REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	14.Oct.98	THESIS

4. TITLE AND SUBTITLE

5. FUNDING NUMBERS

6. AUTHOR(S)

2D LT TILBURY CHAD A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NORTHEASTERN UNIVERSITY

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

98-095

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

AN EFFICIENT, WAVELET BASED, VARIABLE STRUCTURE NETWORK

Chad Allen Tilbury

Submitted to Northeastern University,
College of Computer Science June, 1998,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Computer Science

Abstract

Learning systems, specifically neural networks trained using supervised learning, have become widely accepted and in many cases provide good approximations to the given target function. One of the parameters that tends to make a large difference in the output of the network is its size, or number of units that comprise the network. However, there is often no easy way to determine how large a network should be. A way to combat this problem is through a variable structure algorithm.

Variable structure means that in the course of learning the data, the network also learns the best way to organize itself to represent the data. Thus the number of units and the structure of those units are driven by the characteristics of the data. Wavelets have been found to provide several properties that make the construction of a variable structure algorithm approachable.

Wavelets exhibit good localization in both the spatial and frequency domains. The class of wavelets explored in this thesis is from the family of orthonormal wavelets. Orthonormality implies that there is no redundancy in the information stored by these wavelets. This creates basis units that can be added or removed without affecting their counterparts, lending themselves to additive types of variable structure. It also means that the data is stored in an efficient manner.

Using orthonormal wavelets and a Recursive Least Squares based training algorithm allows for the creation of a hierarchical, multiresolution network that facilitates variable structure. The challenge of this thesis was to create a variable structure network based on the orthonormal Haar wavelet which maintains the above properties while being fast and efficient.

Technical Supervisor:	Dean E. Cerrato
Title:	Senior Member of Technical Staff, Draper Laboratory
Thesis Supervisor:	Ronald J. Williams
Title:	Professor of Computer Science

AN EFFICIENT, WAVELET BASED, VARIABLE STRUCTURE NETWORK

Chad Allen Tilbury

B.S. Computer Science

United States Air Force Academy

(1996)

Submitted to the College of Computer Science

in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
at
NORTHEASTERN UNIVERSITY

June 1998

Signature of Author Chad Allen Tilbury
College of Computer Science
June 1998

Approved by Dean E. Cerrato
Dean E. Cerrato
Technical Supervisor, Charles Stark Draper Laboratory

Certified by Ronald J. Williams
Ronald J. Williams
Associate Professor of Computer Science

AN EFFICIENT, WAVELET BASED, VARIABLE STRUCTURE NETWORK

Chad Allen Tilbury

Submitted to Northeastern University,
College of Computer Science June, 1998,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Computer Science

Abstract

Learning systems, specifically neural networks trained using supervised learning, have become widely accepted and in many cases provide good approximations to the given target function. One of the parameters that tends to make a large difference in the output of the network is its size, or number of units that comprise the network. However, there is often no easy way to determine how large a network should be. A way to combat this problem is through a variable structure algorithm.

Variable structure means that in the course of learning the data, the network also learns the best way to organize itself to represent the data. Thus the number of units and the structure of those units are driven by the characteristics of the data. Wavelets have been found to provide several properties that make the construction of a variable structure algorithm approachable.

Wavelets exhibit good localization in both the spatial and frequency domains. The class of wavelets explored in this thesis is from the family of orthonormal wavelets. Orthonormality implies that there is no redundancy in the information stored by these wavelets. This creates basis units that can be added or removed without affecting their counterparts, lending themselves to additive types of variable structure. It also means that the data is stored in an efficient manner.

Using orthonormal wavelets and a Recursive Least Squares based training algorithm allows for the creation of a hierarchical, multiresolution network that facilitates variable structure. The challenge of this thesis was to create a variable structure network based on the orthonormal Haar wavelet which maintains the above properties while being fast and efficient.

Technical Supervisor:	Dean E. Cerrato
Title:	Senior Member of Technical Staff, Draper Laboratory
Thesis Supervisor:	Ronald J. Williams
Title:	Professor of Computer Science

Acknowledgments

Well let's see. Where shall I begin? I always thought that this would be a fun page to write, but at this point any kind of writing whatsoever is just drudgery. However, I do know that this page will be significantly less cynical than the last acknowledgments I wrote two years ago. Maybe I've changed, but I like to think that I have just enjoyed myself a hell of a lot more.

The first thanks needs to go out to my progenitors. See guys, I increased my vocabulary in Boston! Mom and Dad, thank you for all you have done for me these last two years. Although you had no real input in where the Air Force would send me, I know how much it pained you to have your only son on a distant shore. It must have been even worse that he fell in love with the dreaded East Coast!! Well as luck would have it, D.C. is no closer to home. It does however have a higher crime rate so at least that will keep life interesting. Grandma, Chuck and Carolyn, thanks for coming out to visit and at least trying to understand what I found so charming about this city with the suicidal drivers and out of this world prices.

Next come the friends which really made my entire Boston experience. Although I never saw you at work (I think I made it to one lunch!), we had enough good times outside of Draper to keep things sane and worthwhile. Besides, what fun is beer when you have to go back to work in thirty minutes?? Tony "don't make me fight you" Giustino and Ted "I will never meet my wife in a bar" Conklin-- was it everything we dreamt of? Can you believe that we all managed to make it here? It was great living with you guys, but I tell you someone was dirtying those dishes!! Just like the phantom crapper in London, someone had to do it!! Its hard to believe that we spent two years in a coastal town without ever getting out on a real sailing vessel, but there will be plenty of time (and money) for that. Ted, you can come play on our boat if the wife signs a permission slip!! Chris "postprandial" Dever - well we never did buy those motorcycles, but at least we are still alive. Beau "Shoot the brain" Lintereur, Rudy "no keg stands while I'm driving" Boehmer, Gordon "tell me about older women" Maas, Corey "where are the chicks, eh?" and Dave "man that snow is packed!" - you guys were a blast. Whether it was skiing, camping, hiking, or just imitating the Irish, I think that we really lived it up. Did anyone ever meet those nine girls in the penthouse above us?? Speaking of girls - Nancy "the total package" Risch, I am thankful for every day we got to spend together. Who would have ever thought that love was real?? Sorry I slowed down and got boring in the last months- but honey, I'm getting old! You should all notice that last names were included above in case any of you MIT guys make it big. I deserve to be at least famous by association. Don't forget to support your old

buddy in his old age. There are very few wealthy bartenders in the Caribbean!!

Now on to the boss men. Dean, thanks for being an incredible advisor. You truly let me find my own way through this thesis thing. You forced me to learn, even when I took three times as long to do it!! You were a big part of my learning experience here at Draper and I thank you for that. Ron, it was truly an honor to work under you the last two years. I came to Northeastern because you were there, and I will leave much more enlightened due to the time you spent with me.

Finally, I would like to thank Draper Lab for giving me this amazing opportunity.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Independent Research & Development (IR&D) DFY97 Project #814: Learning Toolbox Development and DFY98 Project #927: Autonomous Systems. Draper Laboratory's generous financial support for this project is greatly appreciated.

Table of Contents

1 Introduction

1.1 Problem Statement

Learning systems, specifically neural networks trained using supervised learning, have become widely accepted and in many cases they provide good approximations for the given input. However, neural networks are still often seen as a "black box" which contains many variables that must be manipulated to achieve the right conditions for good output. To make matters worse, small deviations in the type or degree of the data may call for an entirely new set of parameters for acceptable output.

One of the parameters that tends to make a large difference in the output of the network is the size, or number of units that comprise the network. In multi-layer perceptron architectures this factor decides how much representational power the network has, including the maximum state space dimension that can be represented. In other types of networks such as radial basis and basis/influence networks, the number of units limit the portion of the state space that can be learned and the granularity of the state space representation. One way to eliminate this problem is to ensure that the network has more than enough units to represent the state space. How can this be determined? Heuristics can be formed based on various factors of the input data, but then we are back to an *ad hoc* solution that we cannot be sure is optimal. Additionally, if it does prove

to be correct, how can we be sure that it is efficient? A large number of unnecessary units can seriously affect performance and in some cases, could lead to memorization and bad generalization of the data.

An intelligent way to combat this problem of network size is through a variable structure algorithm. Variable structure basically means that in the course of learning the data, the network also learns the best way to organize itself to represent the data. Thus the number of units and the structure of those units will be driven by the characteristics of the data. Variable structure needs to be differentiated from self-organizing structure. Self-organization models, such as Kohonen networks and others that use competitive learning have the ability to alter their structure according to the data received [8]. The difference is that they still have an initial network size and only have the faculties to change what they are originally given. They do not have the means to add new units as required by the data. Thus a variable structure algorithm has the potential to start from nothing and build the correct network size and structure for the given data.

Wavelets provide several properties that make the construction of a variable structure algorithm approachable. Wavelets are not new in the scientific community but they are relatively new to the field of artificial intelligence and especially neural networks. Wavelets allow good localization in both the spatial and frequency domains meaning that they

are useful in both local and global prediction. The class of wavelets explored in this thesis are from the family of orthonormal wavelets. Orthonormality implies that there is no redundancy in the information stored by these wavelets. This creates stand-alone basis units that can be added or removed without affecting their counterparts, lending themselves to additive types of variable structure. It also means that the data is stored in an efficient manner.

Since wavelets have such a simple construction, the available training methods are diverse. They range from backpropagation to Least Squares methods. This thesis uses the method of Recursive Least Squares (RLS) to train units. RLS offers an optimal solution for the data it has seen. It also offers nice properties like delayed computation, on-line training, easy overlapping of units, and it only requires one pass through the data.

Using orthonormal wavelets and RLS together allows for the creation of a hierarchical, multiresolution network that facilitates variable structure. Of course, there is a tradeoff for the additional feature of variable structure. It comes in the form of more algorithm complexity. Performance drops as the algorithm is forced to keep track of more data and learn things other than the data state space. To be useful the performance loss needs to be minimal. Therefore the challenge of this thesis is to create a variable structure network based on orthonormal wavelets

which maintains the above properties while being fast and efficient.

The efficiency and performance of this variable structure wavelet network will be judged through representative multi-dimensional problems.

1.2 Thesis Overview

This thesis begins with background information in Chapter 2. The information provided is presented as a tutorial on the concepts that the algorithm will employ in subsequent chapters. Section 2.1 starts with simple wavelets, transitions to orthogonal wavelets and then gives specific examples. A multiresolution structure is introduced in Section 2.2 and its advantages are discussed. Next, the options for network training algorithms are presented and critiqued. Finally, Section 2.4 looks at how other researchers have put together learning networks with the above components.

Chapters 3 and 4 constitute the real "meat" of the thesis by presenting our variable structure algorithm. In Chapter 3, we present a very basic algorithm consisting of many of the items discussed in the previous chapter. Section 3.1 starts with an introductory, one-dimensional algorithm that can be built upon in later sections. The next Section modifies the simple algorithm to implement a basic form of variable structure. Finally, Section 3.3 looks at a variety of enhancements that can be added to the simple variable

structure algorithm to both optimize it and tailor it to the needs of the user.

Our one-dimensional network is expanded in Chapter 4. Section 4.1 makes the requisite modifications to allow the network to be used with multiple inputs. Multiple outputs are added in Section 4.2 and two different methods for dealing with them are presented. The next Section delves into the "curse of dimensionality" and talks about the disadvantages of our network in multi-dimension space. The final section in Chapter 4 shows some multi-dimensional results.

At the conclusion of Chapter 4 our variable structure algorithm has been presented in its entirety. Thus far only simple approximations necessary to explain the concepts have been shown. The next step is to prove its usefulness on some real applications and see what the results are. Chapter 5 discusses these experiments in detail. Finally, Chapter 6 summarizes the work in this thesis and gives recommendations for future research.

2 Background

2.1 Wavelets

Wavelets are the foundation of the work in this thesis, so it is necessary to talk about them in some detail. *A wavelet is a local mathematical function that can represent data according to the components of its frequency.* They are represented with the notation $\Psi_{ab}(x)$, with a and b being coefficients, and x being the independent variable. Wavelets began as pure mathematical tools but now they have been absorbed into many disciplines and are being used for everything from human vision and image analysis to fingerprint compression [6]. My work takes place in the context of using wavelet basis functions to approximate a function in a variable structure neural network. I chose to use wavelets because their attributes mesh nicely with this type of algorithm.

There are a large number of different wavelet functions. Some categories are smooth wavelets, orthogonal wavelets, wavelets with compact support, etc. However, some attributes are common to all varieties. Wavelets are unique because they provide good approximating capability in both space and frequency. This is due to their ability to change their shape and size according to the application. In addition to this, many wavelets are easy to create and modify, making them very easy to use. It has also been shown that wavelets can represent many types of functions much more efficiently than other methods such as Fourier analysis [11]. This can

be even further improved by choosing wavelets particularly suited for the type of function being represented, such as smooth, continuous wavelets for a smooth function. Pruning, or thresholding wavelets have also been used to give good, concise, representations. Of course, individual types of wavelets provide their own benefits as well.

2.1.1 General Wavelets

The simplest wavelet available is any function that is half above and half below the input plane (has a zero mean). Stated mathematically:

$$\int \Psi_{ab}(x)dx = 0 \qquad (2.1)$$

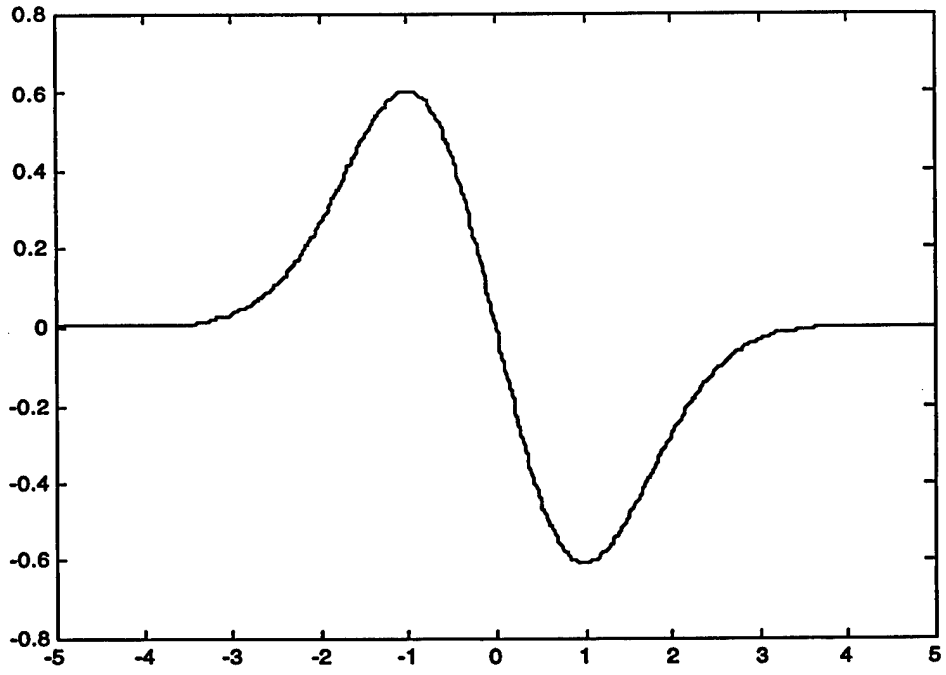


Figure 2.1 A Sample Wavelet Function, $\Psi = -xe^{-x^2/2}$

Wavelets are typically represented in terms of a *Mother Wavelet* [5],

$$\Psi_{ab}(x) \equiv |a|^{-1/2}\Psi\left(\frac{x-b}{a}\right) \qquad (2.2)$$

where

a = dilation coefficient

b = translation coefficient

The dilation coefficient, a, indicates how much the wavelet is compressed or stretched. This determines the size of the wavelet's support. The translation coefficient, b, displaces the wavelet to the desired position. By varying these coefficients, an infinite set of basis functions can be generated from which to construct more complex functions. Due to this, (2.2) is coined the *Mother Wavelet*.

Although we are using a simple wavelet, the transform for representing a function with wavelets, i.e. computing coefficients for the basis functions, is not necessarily simple. One way of doing this is the *Continuous Wavelet Transform*:

$$\text{coeff}(a, b) = \int f(x)\Psi_{ab}(x)dx \qquad (2.3)$$

where

f(x) = function to approximate

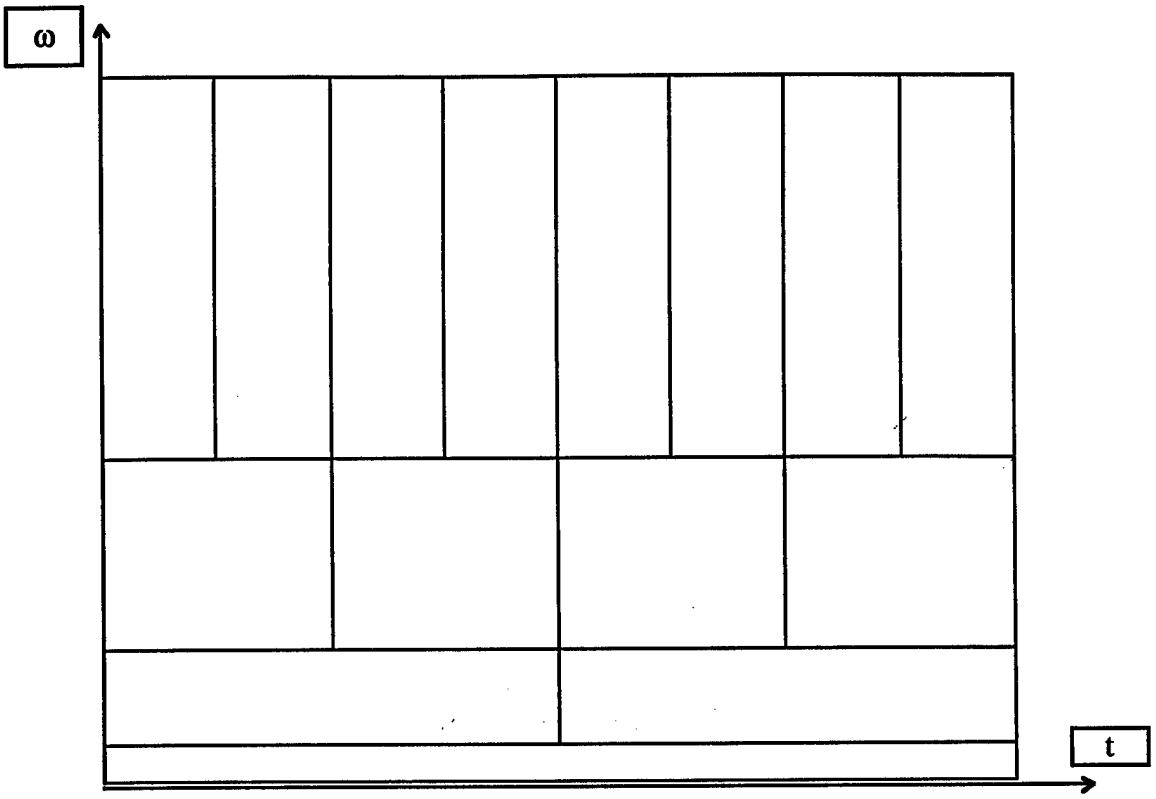
Looking at (2.3), we see that the input function is being projected onto every instance of the Mother Wavelet. This gives us an infinite number of wavelet coefficients. Of course much of the information recorded by these coefficients

is redundant since each wavelet is only infinitesimally different from its neighbors. This oversampling is necessary because there is no way to determine the dependencies of simple wavelets. Therefore, every combination must be tried. This is the opposite of an orthogonal transform, where there are no dependencies.

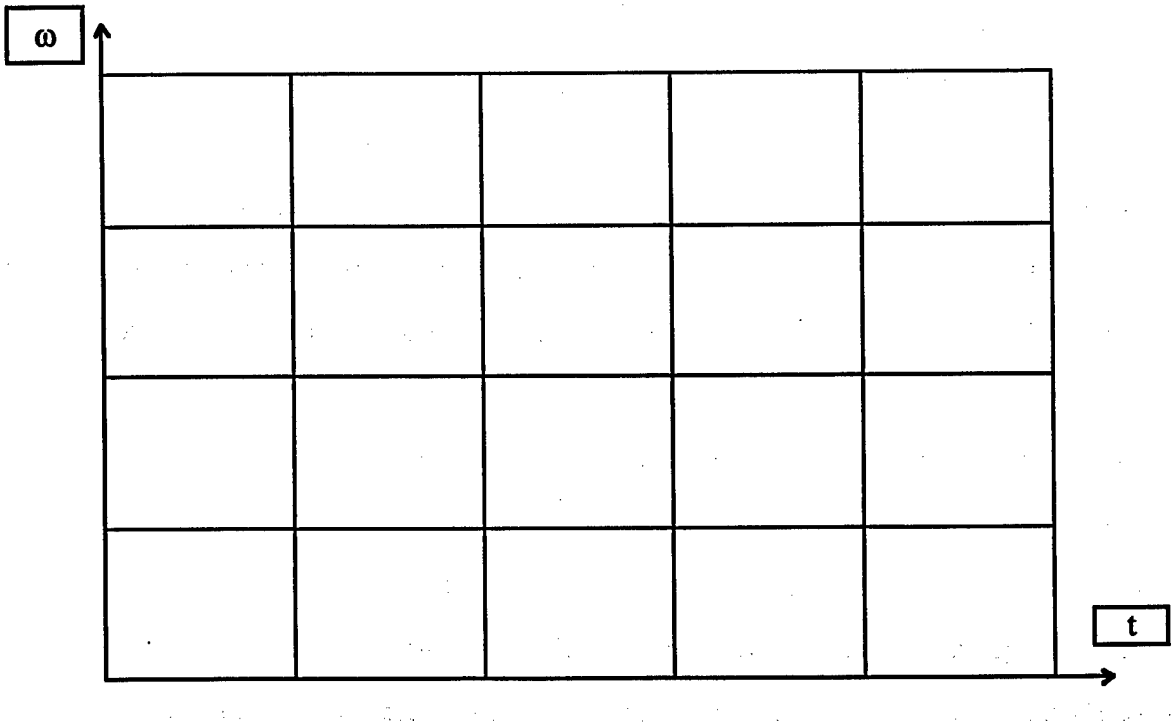
The continuous wavelet transform gives a perfect reconstruction of the input function, but it is only useful in a mathematical sense. In practical applications it is often discretized, trading increased efficiency for a lower quality function approximation. It is often used for recognizing signal characteristics, but it is still much too slow and inefficient for a variable structure algorithm.

Since Fourier Analysis is a more familiar approach, it might be helpful to point out the similarities and differences between the two methods. First, both use basis functions to approximate. Wavelets are analogous to the sines and cosines of the Fourier transform and the Fourier series can be compared to the wavelet coefficients. Also, both transforms are linear and take into account both time and frequency. However, it is the differences that make wavelets desirable. Equation 2.4 contrasts the equation for a Windowed Fourier Transform with that of a Wavelet Transform [9].

$$g^{\omega,\tau}(x) \equiv g(x-\tau)e^{-i\omega\tau} \quad \text{vs.} \quad \Psi_{ab}(x) \equiv |a|^{-1/2}\Psi\left(\frac{x-b}{a}\right) \tag{2.4}$$



Wavelet Transform

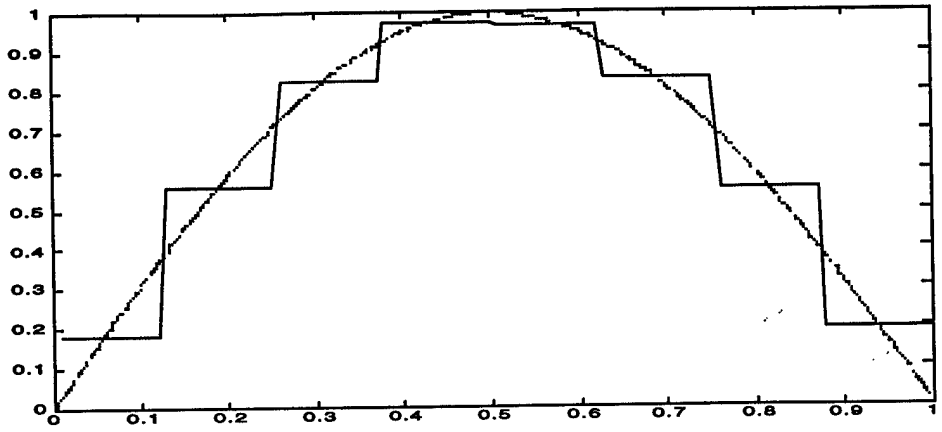


Windowed Fourier Transform

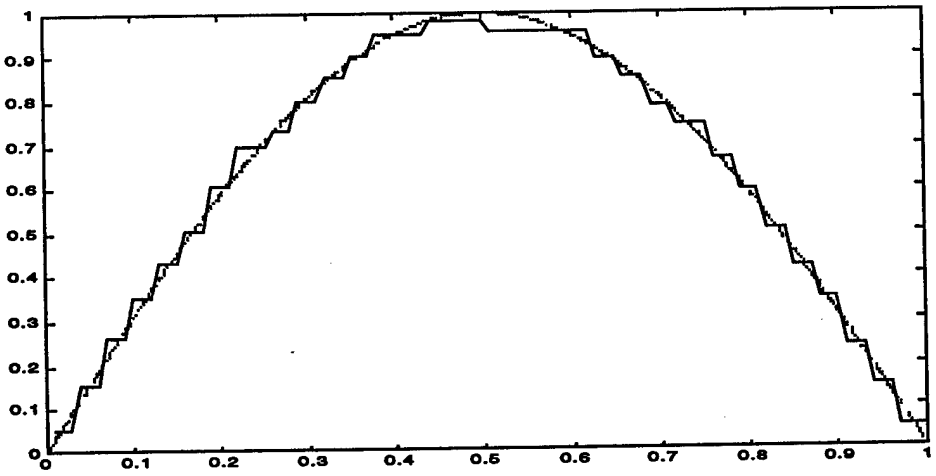
Figure 2.2 Wavelet vs. Windowed Fourier Transform

Figure 2.2 is the graphical depiction of Equation 2.4 with each box being an individual basis function. It shows the overriding difference between the two transforms: the basis functions for the Windowed Fourier Transform remain constant in size with a changing frequency, ω, while the scale frequency, a, of the wavelet basis functions controls the size of the function.

The support of a Fourier analysis is global and unbounded. This indicates that it will be less effective at approximating discontinuities or sharp spikes, etc. Wavelets are localized in time which allow them to excel in these types of functions by creating small, high frequency wavelets to deal with them. Understandably this results in much fewer basis functions than in the Fourier case. This quality is what makes wavelets useful in compression and removing noise in functions. Wavelets are equally advantageous for lower frequency data since their ability to dilate also makes them local in frequency. By utilizing both of these advantages, a wavelet based system can approximate by using small, high frequency wavelets for local features and large, low frequency wavelets for the more global features. Figure 2.3 shows a function approximated using relatively low frequency wavelets and then approximated again with the addition of high frequency wavelets. It is clear that the low frequency wavelets provide a general view of the function while the



Maximum Scale = 3



Maximum Scale = 5

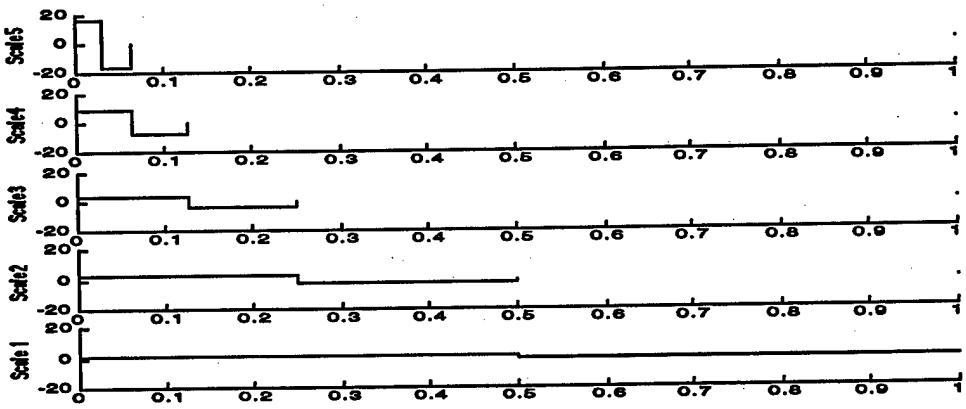


Figure 2.3 Approximation of Sin(πx)

additional high frequency wavelets flesh out the approximation.

2.1.2 Orthogonal Wavelets

The discovery of orthogonal wavelets is a main reason that wavelets are so prominent today. In this case, two wavelets are orthogonal if their inner product is equal to zero. In equation form:

$$\langle \Psi_{ab}(x),\ \Psi_{a',b'}(x) \rangle \equiv \int \Psi_{ab}(x)\ \Psi_{a',b'}(x)\ dx = 0 \qquad (2.5)$$

where

$a \neq a'$ or $b \neq b'$

Orthogonal wavelets also have another orthogonal function associated with them called the father, or scaling function. The name is misleading since the function can be derived from the mother wavelet, but the scaling function provides the wavelet functions with a bias value to be used during approximation. Remembering that the definition of wavelets implies that they have a zero mean, it is apparent why a scaling function is necessary for approximation. Scaling functions will be covered in more detail in section 2.2. The wavelet and scaling functions define an orthogonal basis, meaning that any function can be composed of a linear combination of these orthogonal basis functions [11].

Early in the development of wavelet theory, the mathematics of wavelets were understood, but their application was extremely slow and inefficient. Orthogonal wavelets remedied this because the computation of the

coefficient for any wavelet is independent of any other computations. Essentially, each basis function can be decoupled from the rest. The mapping between the inputs and an orthogonal basis is simple. If we assume a one input, one output function to be learned, the mapping is:

$$f(x) = \Sigma_i \, c_i \Theta_i(x) \qquad (2.6)$$

where

f = output

i = wavelet number

c_i = wavelet coefficient (unknown)

Θ_i = wavelet evaluated at the input

This equation indicates that the function is equal to the sum of the wavelets at the given points, multiplied by their weights. In other words, this is the definition of a basis function approximation. Assuming *m* input/output points and *n* wavelets, we can put the right hand side into matrix form by making a vector of coefficients of size *n* and a matrix of size *m x n* in which every input is evaluated at every wavelet. Using this new matrix form we get:

$$\begin{pmatrix} f(x_1) \\ \vdots \\ f(x_m) \end{pmatrix} = \begin{pmatrix} \Theta_1(x_1) & \cdots & \Theta_n(x_1) \\ \vdots & \ddots & \vdots \\ \Theta_1(x_m) & \cdots & \Theta_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \quad \text{or} \quad f(\underline{x}) = A\underline{c} \qquad (2.7)$$

Now all that is necessary is to solve for c.

$$\underline{c} = A^{-1} f(\underline{x}) \qquad (2.8)$$

Since A will not always be invertible, we need a generalized inverse giving us:

$$\underline{c} = A^{+}f(\underline{x}) \text{ or } (A^{T}A)^{-1}A^{T} \text{ (using Least Squares)} \quad (2.9)$$

This gives us the optimal coefficients for all n wavelets being used to approximate f(x) assuming a large amount of regularly spaced input [1]. It is apparent that this is much easier than the continuous wavelet transform. Of course, for a large amount of data and a large number of wavelets, this approach becomes unworkable because of the size of the components. Techniques such as multiresolution (covered in Section 2.3) must then be used to break the computations up without sacrificing accuracy.

In a neural network structure, orthogonality means that we can have autonomous units that can be added or removed without affecting the other units. By being independent, these wavelets also have the advantage of allowing no redundancy in the storage of information. This contrasts with the continuous wavelet transform which required infinite redundancy. While the lack of redundancy is not desired in some disciplines, it is a great boon for the work in this thesis since we are looking for an efficient network structure. There are a plethora of orthogonal wavelets to choose from. The two used in this thesis are the Haar wavelet and the Mexican Hat wavelet.

2.1.2.1 The Haar Wavelet

The Haar wavelet is the simplest orthogonal wavelet. It is a piece-wise constant function that is defined as having the value 1 on the half-open interval [0,1/2) and the value -1 on the half-open interval [1/2, 1).

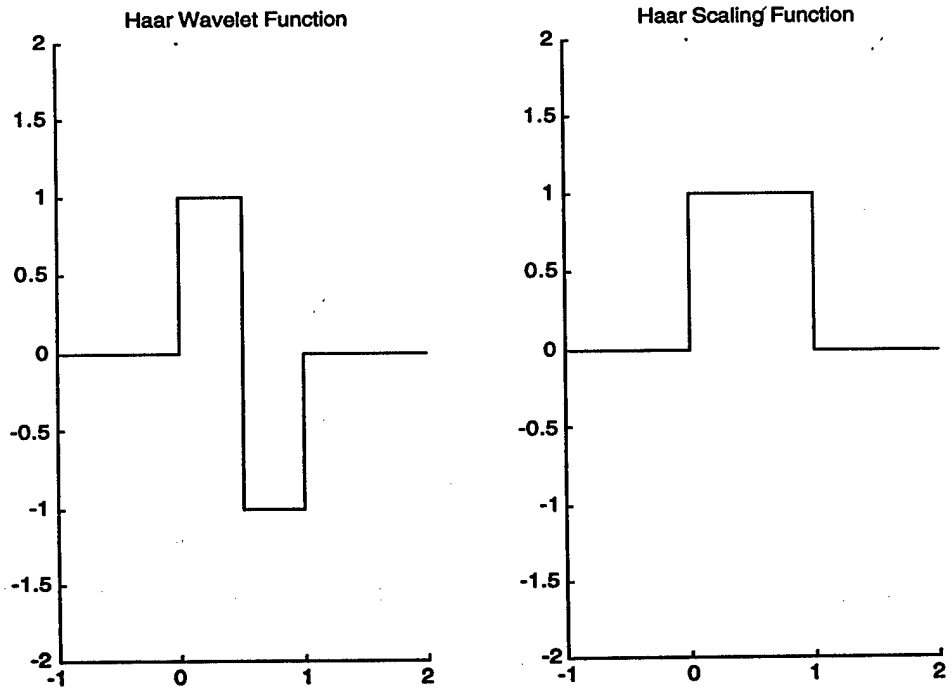


Figure 2.4 The Haar Wavelet and Scaling Function

It has been proven that any continuous function can be approximated to arbitrary accuracy by Haar wavelets given an unlimited number of wavelets of varying locations and frequencies [11].

Haar wavelets are the most computationally efficient of the orthogonal wavelets to date. Since they are a square function, it is easy to compute their values at the inputs.

The Haar scaling function is simple, being defined as having the value 1 on the half-open interval [0,1). Finally, Haar wavelets have compact support. Thus they do not require overlap as many other wavelets do. This results in a large savings of time and space since interactions between overlapping wavelets do not have to be computed.
In order to prevent confusion, it needs to be said that interactions between wavelets can occur while still retaining orthogonality. Wavelet orthogonality simply indicates that if you remove an orthogonal wavelet, information is lost that no other wavelet can compensate for. Therefore, the coefficients of the remaining wavelets will be unchanged.

Even though the Haar wavelet can approximate any continuous function, many researchers have found it not to be very natural in this mode [11]. Instead, they prefer to use smooth basis functions that may be able to approximate the smoothness of the function as well as the actual values. The Mexican Hat wavelet is one of these and is covered in the next section. Often the Haar function is relegated to Boolean type functions. In my work I have found the piecewise-constant Haar wavelet to approximate smooth functions almost as well as smooth basis functions. This fact coupled with the impressive savings in computation time makes Haar wavelets my first choice of basis function in this thesis.

2.1.2.2 The Mexican Hat Wavelet

The Mexican Hat function gets its name from it shape. It was discovered by the field of vision analysis and it is still in use today. The wavelet equation is

$$\Psi(x) = \frac{2}{\sqrt{3}}\pi^{-1/4}(1-x^2)e^{-x^2/2} \qquad (2.10).$$

This equation is the second derivative of the Gaussian function, $e^{-x^2/2}$, normalized so that its L^2 norm is equal to 1 [4].

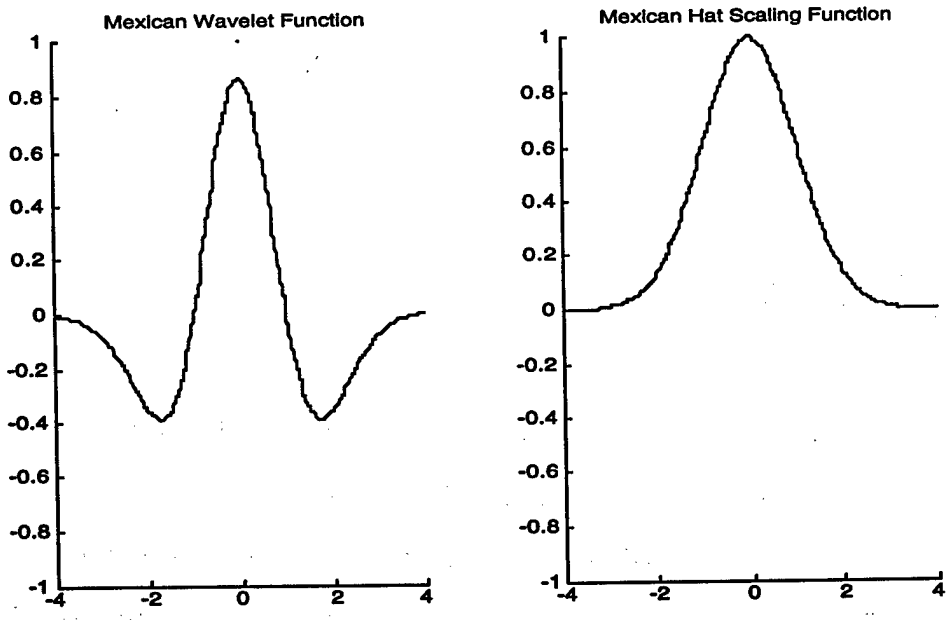


Figure 2.5 The Mexican Hat Wavelet and Scaling Function

I chose to use the Mexican Hat wavelet for several reasons. First, it is easy to compute. While the equation is somewhat lengthy, it is much more desirable than many other wavelets which must be computed by the iteration method [4]. The ease of computation also applies in the case of the

wavelet's scaling function, which is simply the Gaussian. The Mexican Hat wavelet is smooth, which should allow it to approximate and generalize smooth functions easier [11]. This idea is illustrated in the fact that vision analysis cannot use the Haar wavelet due to artifacts caused by its discontinuities that are left in the reconstruction. Thus it is included in this thesis because it is one of the more computationally efficient wavelets to construct and it should provide a good measure against which to judge the performance of the Haar wavelet. The Mexican Hat is a wavelet function that requires the basis functions to overlap. This results in a significant performance drop when compared to wavelets such as the Haar function. So what we are interested in is if the Mexican Hat approximation is good enough to compensate for this decrease in efficiency.

2.1.3 Orthogonal Wavelets in Multiple Dimensions

So far, we have only been discussing wavelets of one dimension. These wavelets are useful for a one input network, but little else. Since most applications require several input variables, multi-dimensional wavelets are a necessity. Luckily, orthogonal wavelets are easily extended to any dimension.

Extension to a n-dimensional basis begins with the one dimensional mother wavelet and scaling function. The basis is found by taking the tensor products of the n-1 dimensional basis with the one dimensional mother wavelet and scaling

function [5]. As an example, when expanding to two dimensions the resulting equations for the basis will be:

Given $\Psi(x)$, $\Phi(x)$:

	$\Phi(x_2)$	$\Psi(x_2)$
$\Phi(x_1)$	$\Phi(x_1)\Phi(x_2)$	$\Phi(x_1)\Psi(x_2)$
$\Psi(x_1)$	$\Psi(x_1)\Phi(x_2)$	$\Psi(x_1)\Psi(x_2)$

Scaling Function : $\Phi(x_1,x_2) = \Phi(x_1)\Phi(x_2)$ (2.11)

Wavelets:

$$\Psi^1(x_1,x_2) = \Phi(x_1)\Psi(x_2) \quad (2.12)$$

$$\Psi^2(x_1,x_2) = \Psi(x_1)\Phi(x_2)$$

$$\Psi^3(x_1,x_2) = \Psi(x_1)\Psi(x_2)$$

This technique will expand any orthonormal wavelet basis to any dimension. An excellent proof of this can be found in Daubechies' paper [5].

From the method above it follows that the number of wavelets increases exponentially with the dimension. For dimension d, there will be 2^d-1 wavelets and one scaling function. The scaling function is just the tensor product of the *n* original scaling functions while the wavelets are the combinations of *n* products of the scaling and wavelet bases. These additional wavelets correspond to the different orientations of the wavelet in that dimension that are necessary for the basis to completely span the function space. However, this is not as much of a limitation as it appears to be. It is often not necessary for every wavelet in the basis to be used in a specific approximation. A

variable structure algorithm can be employed to only activate the wavelets that are necessary for the approximation. This is often significantly less than the total number of available wavelets.

2.2 Multiresolution

Multiresolution is a decomposition technique which allows a hierarchical representation of a set of inputs. The input is decomposed into different resolutions, each at a different scale, and is represented by the difference of information between each scale. Multiresolution was derived from multiscale algorithms used in machine vision and image processing. These methods analyzed signals at different resolutions taking advantage of the fact that images tend to show different things depending on the resolution they are viewed at. However, the methods were confined to a small number of fields due to their inefficiency. Since there was no way to isolate the information in one resolution from the information in the other resolutions they tended to be very redundant.

Wavelets seemed to be a logical choice for a better implementation of multiresolution since they are adept at recording differences. The dilating characteristics of the wavelets allow some to grow large to adapt to low frequencies and some to contract to capture the high frequencies. As the wavelets get smaller and smaller, they capture the image at smaller and smaller resolutions. Of course, ordinary

wavelets do little for the efficiency issue since they also record redundant information. Orthogonal wavelets were needed to eliminate this redundancy. Once they were discovered it quickly became apparent that the two ideas of orthogonality and multiscale were well suited to each other. The first fast, efficient method for determining a wavelet decomposition of a function soon followed and it was dubbed multiresolution analysis [10].

As we know, orthogonality implies that each wavelet coefficient is completely independent of any other coefficients in the analysis. This means that in any function decomposition, each wavelet encodes a portion of the function that no other wavelet does. Mallat presented his multiresolution algorithm as a sequential progression from the finest details to the coarsest (from the smallest dilated wavelets to the largest) but this is not the only way it can be performed [10]. Since the wavelets only encode specific portions, multiresolution can also be done from coarse to fine, which is much more amenable to implementation within a variable structure network like the one presented in a later chapter.

Wavelets are most adept at representing details, or the changes from one representation to another. In order to allow them to just concentrate on these details, another function is necessary to record the general trends of the function. This entity is the scaling function. The scaling function is basically "chosen to satisfy continuity,

smoothness, and tail requirements" of the chosen wavelet [11]. It basically gives the analysis a place to start out from, or bias, by roughly approximating the given function. An interesting thing about the scaling function is that it can be created from the mother wavelet [6]. So for multiresolution analysis, all that is needed is the choice of one mother wavelet and the rest can be calculated from there. It is easy to see the simplicity that makes this type of analysis so intuitively pleasing.

After the mother wavelet is chosen and the scaling function is determined, the next step in the altered version of multiresolution analysis is to choose how coarse of a representation is necessary. This determines how large the scaling and wavelet functions will be that the representation will start out with. In this step it is easy to see why we want to go from coarse to fine since in Mallat's algorithm it is necessary to choose how *fine* a resolution is necessary which is exactly what a variable structure algorithm is trying to determine! Now the signal is divided into two portions: a smoothed, generalized portion and the remainder of the function which I will call the *details*. The smoothed portion is provided by the scaling function and the wavelets encode the details.

At the next iteration of the algorithm we have a representation of the original function but it may still be rough since the wavelets at that level are large (equal to the coarseness chosen) and thus only the details of that size

were encoded. Therefore it is necessary to add smaller wavelets. The next level of wavelets consists of double the number of wavelets that are half as wide. This allows them to fit into the same area as the previous level's wavelets as shown in Figure 2.6.

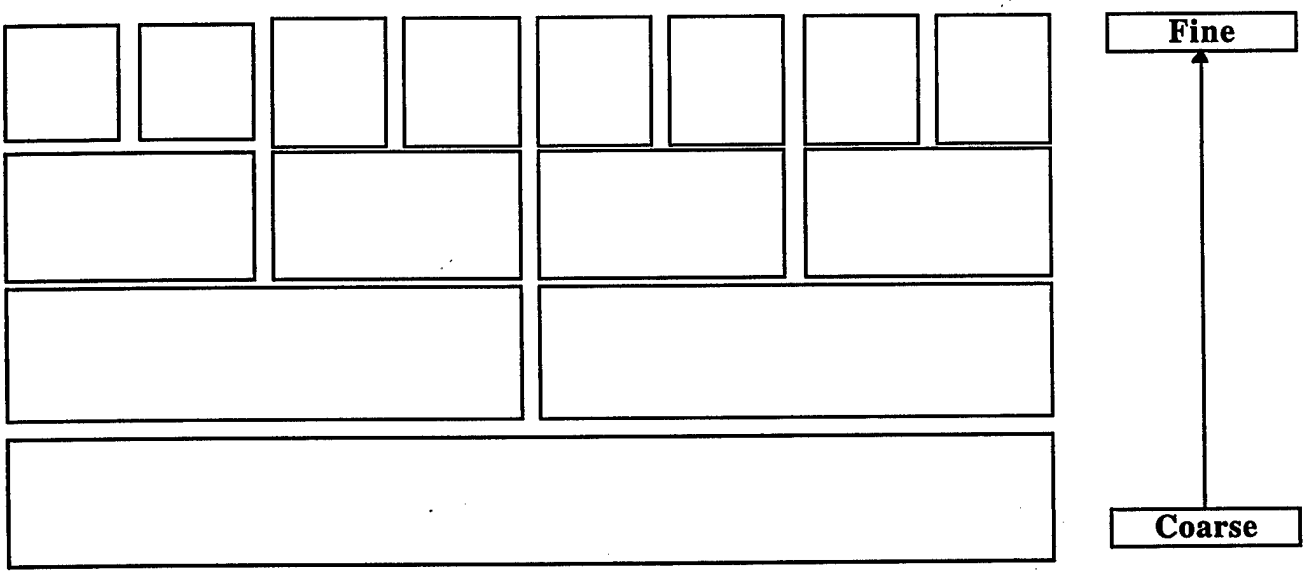


Figure 2.6 Multiresolution Wavelet Analysis

With each successive iteration after the first, the general portion will be the previous approximation (the scaling function plus the wavelets) and the new wavelets will be used to encode as much of the difference between that approximation and the actual function as possible. The process of adding twice as many wavelets at each successive resolution level continues until the function is represented to some specified level. Each successive level of resolution brings the approximation closer to the original function.

The number of wavelets are doubled each time to stay consistent with the Shannon sampling theorem which states that a signal must be sampled at twice its frequency. As the frequency is increased, the number of samples, or wavelet coefficients must be doubled. At the end of the process, we are left with a scaling function coefficient from the coarsest pass and a hierarchy of wavelet coefficients.

Looking at this, we see that the smoothed function could also be represented as the low frequency data of the function and the details could be the high frequency data. This brings about the idea of using filters. The scaling function is a low pass filter and the wavelets constitute various band pass filters. Each level of the multiresolution analysis now has its own pair of filters. Whatever one blocks, the other allows (at that resolution). At each iteration, the frequency of the band pass filter is doubled and convolved with the remainder of the input data from the previous filters, resulting in the new set of coefficients for that resolution. Of course, using filters eliminates the need for wavelets altogether. I don't use this implementation, but it is one more perspective on multiresolution analysis and many have suggested that it could make the technique very fast if implemented in hardware [6].

Multiresolution provides a very good model for variable structure learning using wavelets. It has inherent flexibility in that it does not depend on any specific wavelet. It is also very fast and efficient. When used in

the coarse to fine framework, it provides the best representation of a function possible so far, at any time during the approximation. This is true because each resolution level just adds a little more accuracy to an already approximated function. This can be particularly helpful in time-critical applications where an approximation is given a certain time limit and it must present the best solution possible at that limit. Perhaps the best attribute of this multiresolution method will become apparent in Chapter 3 when the actual variable structure algorithm is discussed. Multiresolution allows orthogonal wavelets to be chosen from any level, reassembled into the desired structure, and trained easily.

2.3 Methods of Training

It is important to note that multiresolution analysis has nothing to do with actually determining the coefficients of the set of wavelet basis functions. Its purpose is to break the network calculations down into a computationally feasible manner that gives the same results as if a full wavelet transform were performed on the data. Multiresolution provides a structure for the wavelets that is independent of the training method. This is not to say that the training method is unimportant. It just plays a different role. Training deals with optimization and data fitting. Given a specific structure, a training algorithm works to optimize the structure's parameters. In the

subsequent sections I have chosen to present the two most common ways to train wavelet networks, Gradient methods and Least Squares, as well as Recursive Least Squares which is employed in this thesis.

2.3.1 Gradient Methods

Gradient methods have been a staple in neural networks since they were applied to multi-layer perceptrons in the 1980's. The best way to understand them is through imagining an error surface like the one in Figure 2.7.

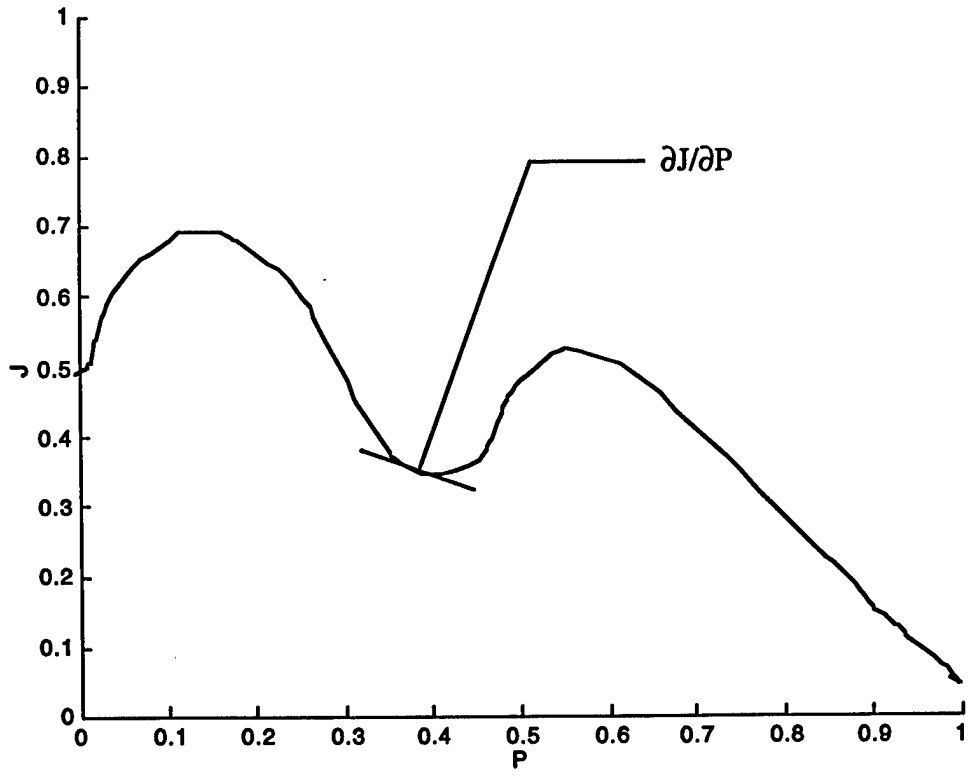


Figure 2.7 Error Surface

If we let J be some measure of the total error and P be a space created by all of the adjustable parameters in the network, then Figure 2.7 is the error as a function of the

parameter values. On this surface there will be some set of parameter values which will give the minimum value for J, or the total error of the network.

Of the many flavors of gradient descent algorithms, Steepest Descent is the simplest and most widely used. *A gradient gives the direction of maximum increase of the function at that point.* It is basically the vector of the partial derivatives of y given x. In our case the gradient is $\partial J/\partial P$ and it is shown on Figure 2.7. A Steepest Descent algorithm attempts to minimize J by finding the negative gradient (since we want the maximum decrease of J at the points P) and adjusting the parameters to go in that direction. Since it uses the negative gradient, a small change in that direction is likely to reduce the current error by the greatest amount.

Backpropagation is a very popular method for computing network parameters using the gradient. It provides us with a nice, general model for looking at how a gradient descent algorithm can be implemented.

Backpropagation basically divides the calculation of the gradient on the error surface into components that each weight is responsible for [13]. The algorithm starts by taking the input vector and querying the current network for an output vector. This output is compared to the desired output, resulting in an error value. If let y^* be the desired output, and y be the actual output, $e = y^* - y$. Usually in

this type of algorithm we are interested in the sum squared error (SSE) which can be represented as $J = \frac{1}{2}\|e\|^2$.

At this point, backpropagation is used to assess the blame for the error. This is done by using partial derivatives along with the chain rule. For example, if our function to approximate is $y(x;p) = f(g(h(x;p)))$ and we want to find the contribution of x to J, the equation would be:

$$\left.\frac{\partial J}{\partial x}\right|_p = \frac{\partial J}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x} \qquad (2.13)$$

Once the we know how much x contributes to the error, it is possible to update the weights pertaining to x. The update rule is:

$$\Delta x = -\alpha\frac{\partial J}{\partial x} \qquad (2.14)$$

where

α is the learning rate (typically small and positive)

The learning rate is used to prevent large jumps along the gradient and allow a smooth traversal of the error surface. Thus backpropagation attempts to follow the gradient by continually changing the network coefficients according to how much of the error they caused.

While gradient algorithms are an elegant way to update parameters, they have their difficulties. One problem is that they have a highly variable convergence time. In general, it is difficult if not impossible to determine how many epochs, or passes through the data set, will be required

for the algorithm to converge to a local minimum. When coupled with the fact that reasonable outputs may not result until convergence, it is easy to see that this may not be the algorithm of choice for time intensive applications. Finally, gradient algorithms have the potential to converge to sub-optimal solutions corresponding to local minima which can be significantly less accurate than the global minimum. Despite these problems, there has been some success achieved using gradient descent methods and wavelets (see Section 2.4). However, these methods are not very amenable to the on-line, variable structure algorithm that I wish to develop.

2.3.2 Least Squares

Least Squares (LS) is a method for solving an overdetermined system of linear equations such that the L^2 norm of the error is minimized. "Overdetermined" means that we have more data than necessary to solve for the coefficients. Assuming m equations with n unknowns, m > n. The Least Squares solution is very basic. If we have a system, $Ax = b$, we want to minimize $\|Ax-b\|^2$ [14]. Solving this for x:

$$x = \left(A^T A\right)^{-1} A^T b \tag{2.15}$$

where

A is m x n

x is n x 1

b is m x 1

The easy method for finding wavelet coefficients in Section 2.1.2 employs Least Squares.

Least Squares has several benefits besides being easy to compute. Given a system of equations, Least Squares will provide the optimal solution. It also allows input data to be weighted according to its reliability. This is done by simply adding a weight matrix into the above equation [14]:

$$x = \left(A^T C A\right)^{-1} A^T C b \tag{2.16}$$

where

C is $W^T W$

W is the weighting matrix containing a value for each error, e = b - Ax

If the errors are independent, W will be a diagonal matrix. Otherwise it will also contain off-diagonal weights. In the LS algorithm discussed before W is just the identity matrix.

On the negative side, Least Squares is not particularly well suited to on-line training. The matrix A is created using the input data and any new data must be added by creating a new row. Thus, with a large amount of data, the matrix gets very large. Another problem is that every time the matrix A is updated, x must be solved for again when all that is needed is the change in x with respect to the new data. These problems are remedied with Recursive Least Squares.

2.3.3 Recursive Least Squares

Recursive Least Squares (RLS) is a way to incrementally solve a system using the Least Squares method. This is done by assuming the first estimate is optimal with respect to the data seen so far. As more data arrives, an update to the previous estimate is made to make the solution optimal for all the data to that point. The information necessary to continue updating is saved in a covariance matrix, eliminating the need to save all previous measurements. This is akin to computing a running average by just saving the sum and the number of values so far. The RLS equations are [14]:

$$P_i^{-1} = P_{i-1}^{-1} + A_i^T V_i^{-1} A_i \tag{2.17}$$

$$x_i = x_{i-1} + K_i(b_i - A_i x_{i-1}) \text{ with } K_i = P_i A_i^T V_i^{-1}$$

where

P = covariance matrix of the input data

V = covariance matrix of the input "noise" or uncertainty

A = information matrix

x = coefficient vector

b = observation, or output, vector

2.3.3.1 Weighting Observations in RLS

Like Least Squares, an important ability with RLS is that observations, or inputs, can be weighted. This is accomplished via the V matrix in Equation 2.17. Assuming every observation is independent, the V matrix will look like:

$$V = \begin{bmatrix} \sigma_1^2 & & & \\ & \sigma_{21}^2 & & \\ & & \ddots & \\ & & & \sigma_n^2 \end{bmatrix} \text{ and } V^{-1} = \begin{bmatrix} \frac{1}{\sigma_1^2} & & & \\ & \frac{1}{\sigma_{21}^2} & & \\ & & \ddots & \\ & & & \frac{1}{\sigma_n^2} \end{bmatrix} \qquad (2.18)$$

Therefore the V matrix could be considered the amount of error, or noise, in each input while the V^{-1} matrix is the weighting that each input will get in the RLS algorithm.

If we take this a little further, we can develop age weighting. In this case, we want each new input to have more weight than any of the previous inputs. If we let λ, $0 \leq \lambda \leq 1$, be the discount factor that each previous input will get relative to the new input we get a V^{-1} matrix like:

$$V^{-1} = \begin{bmatrix} \lambda^n & & & & \\ & \ddots & & & \\ & & \lambda^2 & & \\ & & & \lambda & \\ & & & & 1 \end{bmatrix} \qquad (2.19)$$

Equation 2.19 tells us that every time a new input is added, the previous inputs are reduced again by the discount factor. In an on-line algorithm we don't want to have to deal with this V matrix so we can rewrite the first equation of Equation 2.17 to look like this:

$$P_i^{-1} = \lambda P_{i-1}^{-1} + A_i^T A_i \qquad (2.20)$$

In this recursive equation, the old covariance matrix of the input data is discounted by λ at every iteration. This is equivalent to Equation 2.17 using Equation 2.19 and it provides us with an easy way to perform age weighting on the input data. As a side note, RLS using data weighting is still learning the optimal solution. We have just changed the nature of the solution somewhat. RLS now learns the optimal solution given a particular weighting scheme.

2.3.3.2 Summary of RLS

RLS successfully eliminates many of the limitations of LS. The size of the covariance matrix is constant with respect to the input data in contrast to LS which requires an extra row for every input. When used with streams of data it is significantly more efficient since redundant calculations are avoided; it can solve for just the change in x with respect to the new data instead of solving the entire equation again like Least Squares. Both RLS and LS always give an optimal solution relative to the data they have seen. In contrast to gradient methods, it gives this solution by only making one pass through the data, whereas gradient methods require multiple epochs.

Of course, RLS does have some potential problems as well. For a large number of unknowns, n, the covariance matrix, P, can be very large since the size of P is n x n. Thus in many cases a significant amount of storage space is a necessity. Additionally, the RLS equations require an

inverse of P which can be a large amount of computational work, especially if P is large. The ramifications of these factors can be reduced by breaking up the unknowns into smaller clusters (if independent) and by only calculating x periodically, reducing the number of inverses taken. These solutions along with the other positive attributes of RLS make it a very nice training algorithm for use in an on-line wavelet basis function network.

2.4 Putting It All Together

Before outlining our algorithm, it might be instructive to see how others have constructed wavelet networks.

Zahn and Benveniste were working with wavelet neural networks as early as 1991 [16]. Their work in the area was successful and their papers have been widely distributed. In their network, they use continuous wavelets in a multi-layer network structure trained by a stochastic gradient descent algorithm. These choices drive the rest of their algorithm.

The choice of continuous wavelets is a simple one. Since they are not using a structure which requires orthogonal wavelets (such as multiresolution) they have their pick of whatever wavelet base best fits their needs. Their choice of the multilayer network structure is a little more complicated. It allows the algorithm to be compared to much of the neural network literature which are also constructed of the same multilayer structures. Additionally, by using gradient descent with a nonlinear feedforward network they

are basically performing nonlinear regression. This is well suited for continuous inputs and outputs, performs well with noisy data, and will find the best fit for its associated network structure. In fact, Cybenko proves that a multi-layer network can represent any continuous function, if it has the correct number of hidden units [3].

The main drawback with this approach, and specifically with multi-layer networks, is that choosing the number of hidden units is not an easy problem. Too many units result in bad generalization due to memorization of the data. Too few units can cause bad approximation since the network may not have the representational power needed to accurately approximate the input function. This is a difficult problem which is addressed in this thesis through a variable structure algorithm. Another problem with this structure is the inability to incorporate *a priori* knowledge into this type of algorithm. This is due to the lack of transparency in the weights. Finally, multilayer networks used with gradient descent have the potential to converge to sub-optimal solutions (local minima) when their initial conditions are not set correctly.

To get around many of these problems, Zahn and Beneviste used a complicated initialization phase. They used a fixed structure, but chose the wavelets in this structure by looking at an initial batch of data. This fixed the dilation and translation coefficients, leaving only the weights of the wavelets as adjustable parameters. They used a variety of

constraints on these adjustable parameters to try to prevent convergence to local minima. The algorithm was successful in that it was able to get the same results with significantly fewer wavelets than a strict, fixed wavelet decomposition. However, the computational complexity of the algorithm was high, particularly in the initialization stage. Also, there is no guarantee that the local minima will be avoided. It does have the ability to incorporate *a priori* information into its structure by choosing specific wavelets, but that structure is still fixed. This fixed structure implies that it may have problems adapting to new data that fall outside of the range of the initialization batch.

Bakshi, Koulouris, and Stephanopoulos present an algorithm that is much closer to what this thesis deals with [8, 13]. Their network consists of orthonormal wavelets in a multiresolution structure. They chose the Least Squares method to train their wavelets. Using a multiresolution structure they perform variable structure based on the L^{∞} norm. They augment this variable structure by using the technique of cross-validation, in which the network is tested on previously unseen data and adjusted accordingly. Pruning of wavelet basis functions is used in order to increase generalization.

Overall, Bakshi, Koulouris and Stephanopoulos' algorithm is along the same lines as what this thesis is exploring with a few exceptions. Their algorithm is not set up for on-line learning, limiting its usefulness. This is mainly due to the

cross-validation and pruning used, but also because LS isn't an efficient algorithm for on-line learning. Additionally, much of their success has come about using non-compactly supported orthogonal wavelets such as the Battle-Lemarie and the Mexican Hat wavelet [2]. While these wavelets approximate well, they tend to be much more inefficient than compactly supported orthogonal wavelets such as the Haar and Daubechies wavelets [5]. Finally, their work is almost completely confined to one dimensional wavelets.

The two variations presented above were successes, but they leave much to be improved upon. The purpose of this thesis is to build upon these ideas and create a wavelet based, fast, efficient, on-line variable structure algorithm that can be used in a multi-dimensional input space.

3 A 1-D Wavelet-Based Variable Structure Learning Algorithm

While the purpose of this thesis is to present an algorithm for an *n*-dimensional wavelet-based variable structure algorithm, this section will deal with a simplified version of that concept. We will start with a one input, one output system, and build the design from there. This is a reasonable approach since the elements used in this version will all be applicable to the *n*-dimensional case.

Section 3.1 will begin with a specific structure taken from components described in Chapter 2: a multiresolution structure of Haar wavelets trained by Recursive Least Squares. Specifics are used to simplify the discussion and facilitate the description of the algorithm. Section 3.1.4 will discuss changes to make the algorithm more general. In Section 3.2 we will introduce variable structure elements into the algorithm, creating a bare bones version of the algorithm that will be used for the rest of this paper. Finally, in Section 3.3 enhancements and modifications to the algorithm will be discussed.

3.1 The Basic Algorithm

Now that we have the background information from Chapter 2, it is finally time to put the components together and create a simple wavelet network. Figure 3.1 displays how this network will work.

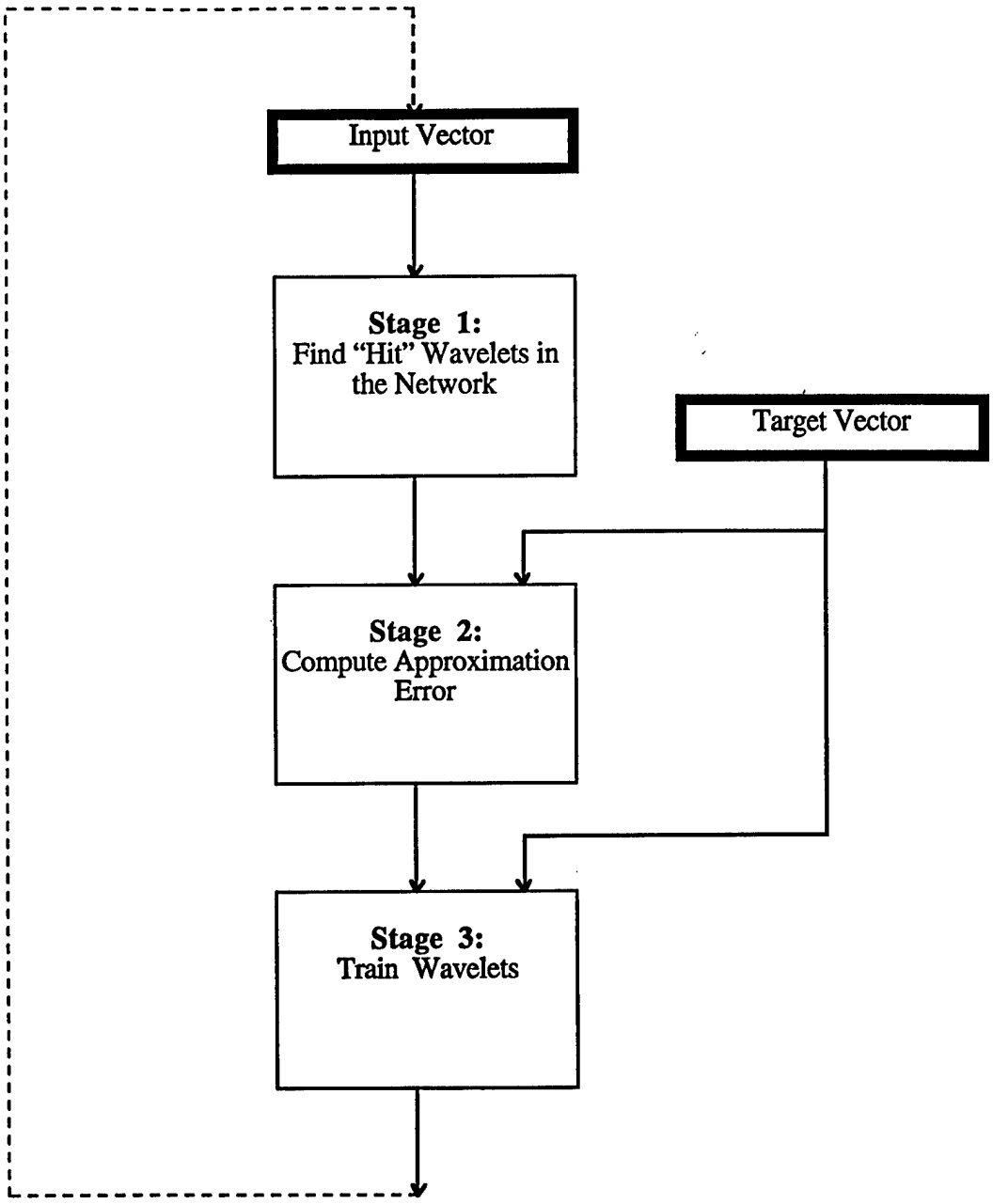


Figure 3.1 Basic Algorithm Flow Chart

Stage one uses the network organization to determine which wavelets are "hit" by the input vector (Section 3.1.1). *A wavelet is hit if the input vector falls within its support.* Stage two looks at the hit wavelets and determines what their

outputs will be for the given input (Section 3.1.2). This effectively gives us the network output for the input vector. We can then compute the error in the approximation. Stage three takes the network output vector and the approximation error and trains the wavelets to better represent the input data (Section 3.1.3).

3.1.1 Stage One: Finding the Hit Wavelets

The goal of this algorithm is to construct the smallest network possible while still approximating the input function. This is done by restricting the choice of wavelets (Haar in this case) to the bare minimum spatial and frequency (dilation and translation) dimensions. Even doing this can still lead to a potentially infinite number of basis functions as the input gets larger in space and finer in frequency. Thus we need some way to organize and keep track of which wavelets will be used for a given approximation. Our algorithm relies on wavelet structures (Section 3.1.1.1), data normalization (Section 3.1.1.2) and multiresolution (Section 3.1.1.3) for this task. These all take place once, before the on-line approximation begins. Once an appropriate wavelet structure is created, it is easy to determine on-line which wavelets will be hit by a given input vector (Section 3.1.1.4).

3.1.1.1 Wavelet Description

The first step in organizing the wavelet bases is to provide a way to access and describe the wavelets that will be used in the approximation. This is done by assuming the wavelets are independent constructs represented by a general structure:

Wavelet:

ID Number	Location	Coefficient

ID Number - An unique numbering used to identify one wavelet from another

Location - A vector containing the unit's center

Coefficient - The wavelet coefficient

This structure tells us the translation of the wavelet (location), gives us a way to identify the wavelet, and keeps the wavelet's training information (in the form of a coefficient). Dilation information is provided by the network structure, specifically by what resolution level it is at.

3.1.1.2 Data Normalization

The first step in culling the available wavelets to a reasonable number is to limit the translation space. In our algorithm this is done through normalizing the input data. Input data are assumed to be normalized in the range of 0 and 1 before they are passed to the algorithm. By separating the

normalization from the algorithm we save the algorithm from storing any of the data, or information about the data. This frees the algorithm to be used for true on-line approximations. Section 5.1.1 discusses the steps necessary to normalize data for a real application.

Normalization significantly reduces the number of wavelets needed to produce an approximation. First, we know the maximum size wavelet available for use. That wavelet is the one which completely spans the input domain (in our case, between 0 and 1). This makes sense because we know that there will be no data outside of that range due to normalization. Subsection 3.1.1.2.1 goes further into this topic.

This type of normalization is not the only solution to the problem of limiting the extent and the number of wavelets. Another method could be to have variable boundaries according to the data seen so far. As new data is introduced which is outside the current range, the boundaries are extended, increasing the number of viable wavelets that can be trained. This is an aspect of variable structure that we chose not to pursue since data normalization was an easy solution. There are other solutions, but for our purposes, the simple type presented above suffices.

3.1.1.2.1 Truncating Wavelets

If we use data normalization we must enforce strict boundaries around the input domain. If a wavelet extends

outside of these boundaries, only the portion of the wavelet within the input domain will be hit by the data. This effectively changes the shape of the wavelet (see Figure 3.2). This new shape will not be in the orthogonal wavelet basis and thus must be discarded.

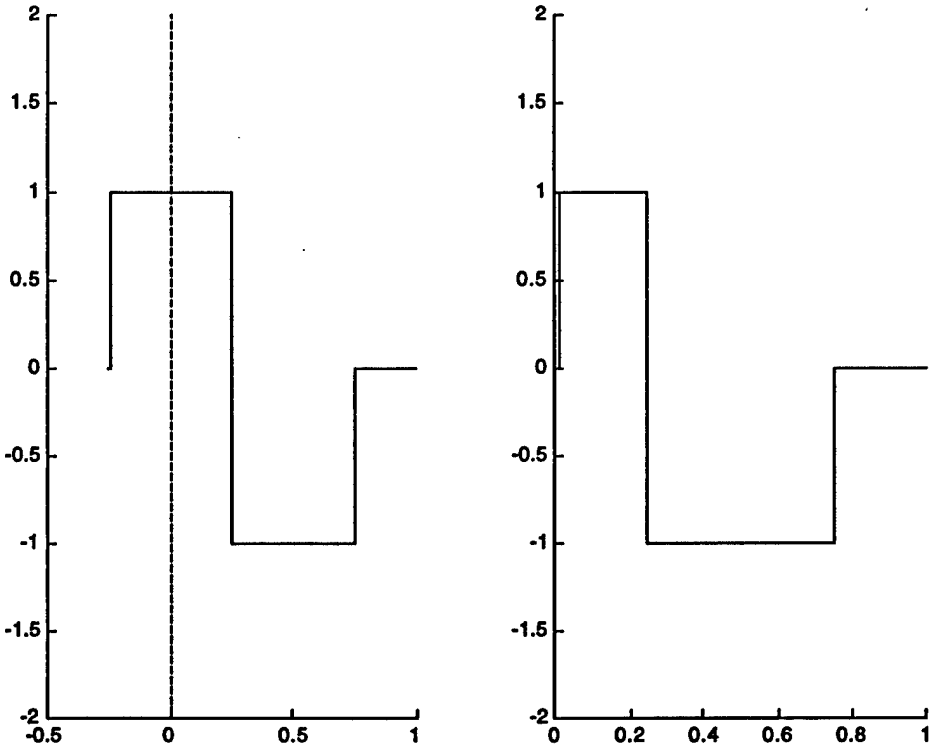


Figure 3.2 A Wavelet Which Extends Past the Normalized Range and its Resulting Shape

It seems prudent to provide a short illustration of why a cutoff wavelet will no longer be orthogonal to the basis. The primary reason for this is because an orthogonal basis assumes that every wavelet used in the basis will be hit by regularly spaced, dense data [10]. Although this is only in the perfect case, a cut off wavelet will <u>never</u> have data hit

the portion that extends beyond the normalized range. This means that we can't even approximate regularly spaced, dense data and thus it can only be orthogonal by chance. We can construct a shorter example by looking at Figure 3.3. The top function is the truncated wavelet from Figure 3.2. The function on the bottom is the Haar scaling function.

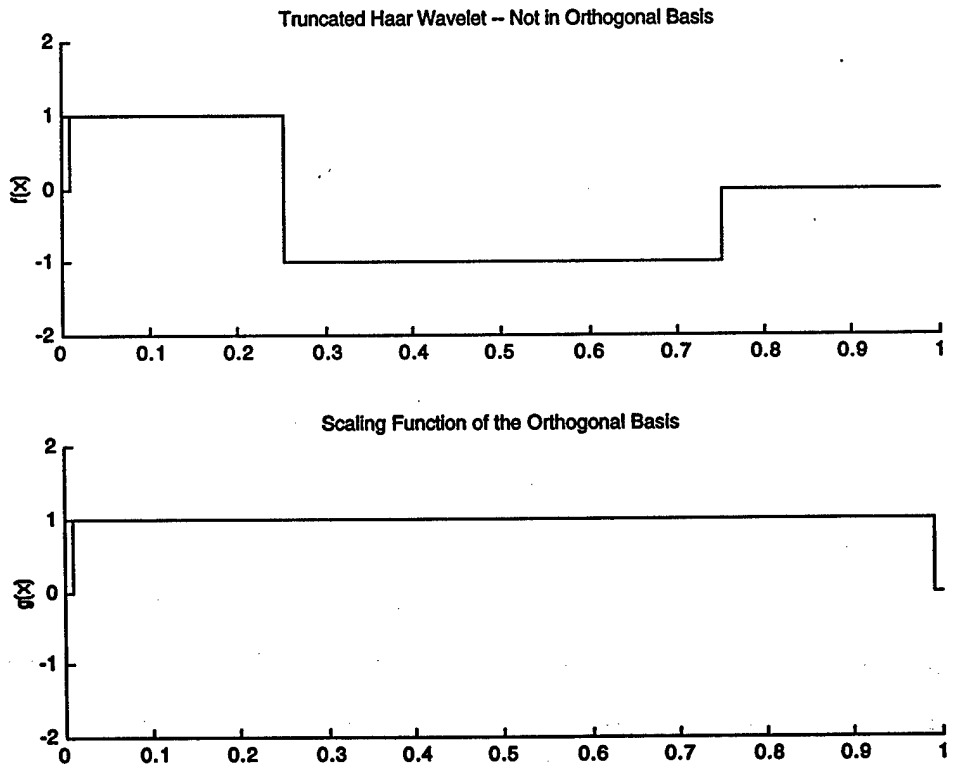


Figure 3.3 Truncated Wavelet Shown with Scaling Function

Equation 2.5 told us that the integral of the product of two orthogonal wavelets must be zero. Taking the integral of the functions in Figure 3.3:

$$\int f(x)g(x) = \int_0^{.25} f(x)g(x) + \int_{.25}^{.75} f(x)g(x) + \int_{.75}^{1} f(x)g(x) \neq 0$$

By using this same principle, we can reduce the number of viable wavelets for the approximation to only those which fit within the input domain.

3.1.1.3 Multiresolution

Normalization tells us if a wavelet is viable, but it does not tell us exactly what wavelets will be necessary. Multiresolution provides us with a hierarchical design that determines where each wavelet will go and what its dilation will be. In our algorithm this process is very simple. The largest wavelet available along with the associated scaling function for the orthonormal basis (reference Section 2.2) are considered to be resolution level one. This wavelet and scaling function will span the input domain. The dilation of level one is set to the value one. On top of this, we are able to construct finer and finer grids of wavelets which tell us the smaller translation steps and the higher dilation levels (i.e. higher frequency) of every wavelet necessary for the approximation.

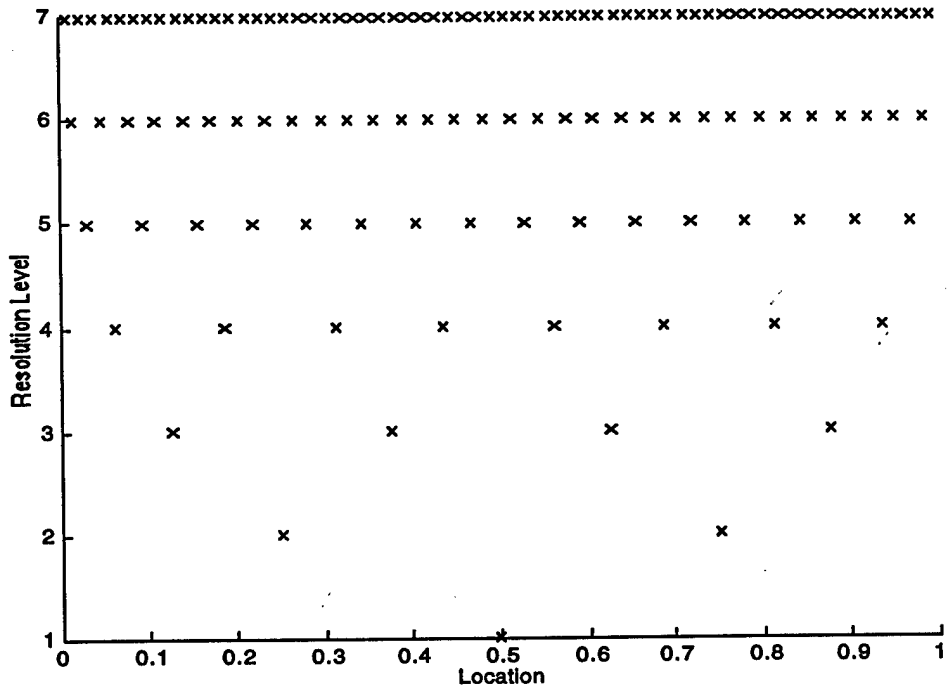


Figure 3.3 Location Grid for Resolution Levels 1-7

At each successive resolution in the grid, we get these relationships:

Frequency (dilation): 2x the next lower Frequency

Spatial (translation): distance apart is reduced by 1/2 from next lower resolution

This allows us to determine how many wavelets are on a particular level (since the range is normalized to 1) and what their locations (translation coefficients) are. If we know what resolution level we want to stop approximating at (Figure 3.3 assumes level 7) we know exactly how many wavelets will potentially be necessary, as well as the translation and dilation coefficients for them all.

3.1.1.4 Finding Wavelet Hits for Compactly Supported Wavelets

This is the part of stage one that takes place on-line. Luckily, determining which wavelets should be trained is really the easy part. This is especially true for wavelets which do not overlap, such as the Haar wavelet. The multiresolution grid structure that we built above basically tiles the space with these wavelets. As seen in Figure 3.4, each wavelet has a specific location range, or support, that it is responsible for. A wavelet will only be trained if the input falls within its range, making this step completely input driven. Since the ranges do not overlap, only one wavelet will be chosen for each of the resolution levels. This indicates that for each input, only n wavelet units will be trained, with n equal to the maximum resolution number. The equation for determining which wavelet in a given resolution level has been hit is:

$$num = \left\lceil {input}/{wavelet\ range} \right\rceil \tag{3.1}$$

where

$\lceil\ \rceil$ is the ceiling function

num is an integer referring to the position of the wavelet within the given resolution level

wavelet range is a value determined by the size of the wavelet at the given resolution level

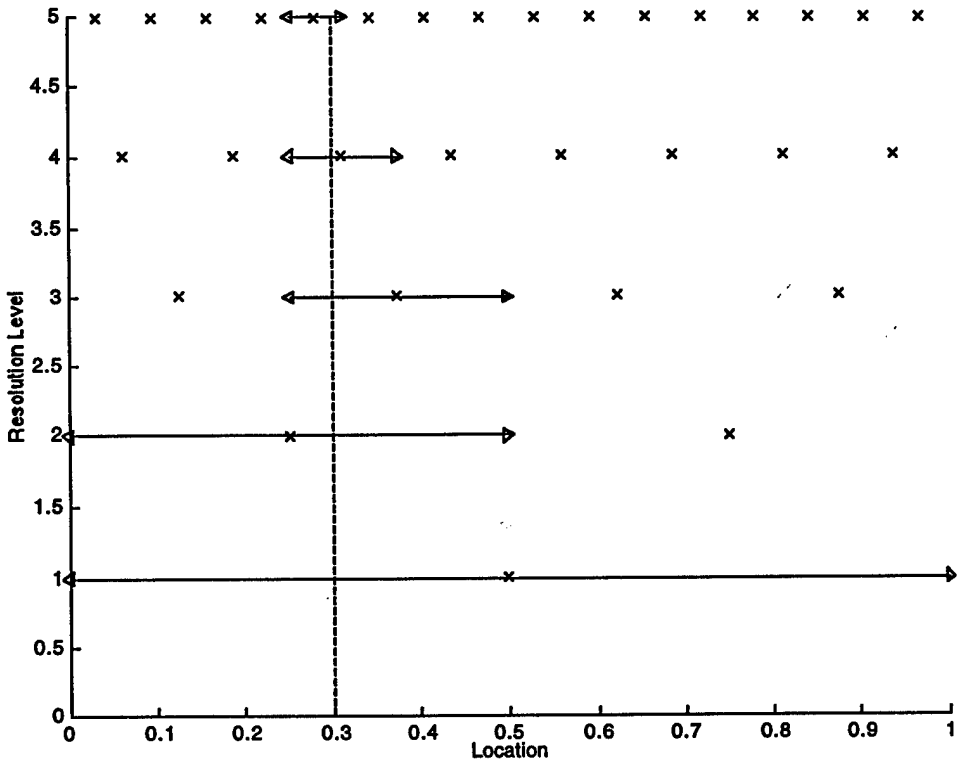


Figure 3.4 Wavelets With Input = .3 Within Their Range

As "hit" wavelets are found they are kept by their identification number to be updated later. Once a wavelet has been found for every resolution level, stage two can begin.

3.1.2 Stage Two: Compute Network Output and Approximation Error

At this stage we assume that we have identified the hit wavelets and scaling functions. Now we must determine what values the wavelets give us for this input vector (the

network approximation) and how that compares to the expected output (the approximation error). The equation to determine the network approximation is:

$$y(\underline{x}) = \underline{a}^T(\underline{x})\underline{c} = \sum_i a_i(\underline{x})c_i \qquad (3.2)$$

where

y is the network approximation

$\underline{a}(\underline{x})$ is the wavelet evaluated at the inputs

$\underline{c}$ is the wavelet coefficients

The network output is the sum of the wavelet basis functions evaluated at the input vector. The wavelet coefficient is taken from the wavelet structure. The value $\underline{a}(\underline{x})$ is a function of the location of the wavelet and the network input. Figure 3.5 shows $\underline{a}(\underline{x})$ being computed on two Haar wavelets. Wavelet 1 gives a value of -1 for $\underline{a}$ at the input while Wavelet 2 gives a value of 1.

Once we have the network's output for the given input the only thing left for this stage to complete is to determine the approximation error. This is simply the squared error between the actual and expected outputs :

$$(\textit{expected output} - \textit{actual output})^2 \qquad (3.3)$$

This value is then passed on to the next stage to aid in training the wavelet units.

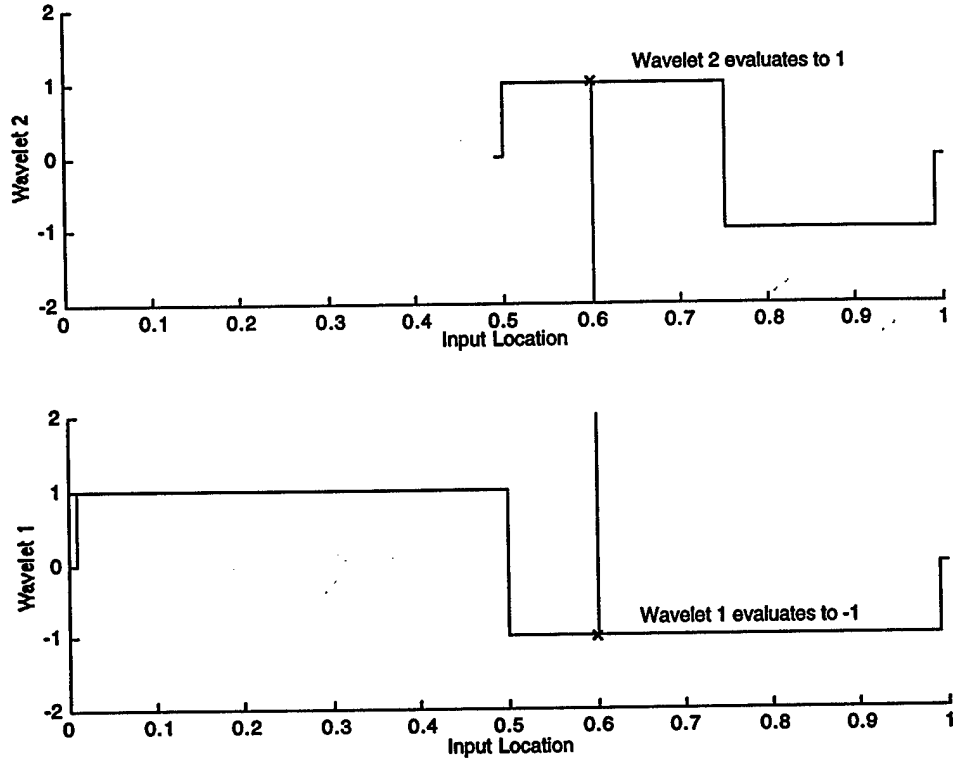


Figure 3.5 Evaluating Two Wavelets in the Grid at Input= .6

3.1.3 Stage Three: Train Wavelets

Training is accomplished in an on-line manner. This allows the algorithm to be used by a much wider selection of problems since on-line training can be simulated by using a batch if necessary. It is also the most suitable method for our applications of interest since we will be looking at on-line dynamical systems where the full scope of the data encountered is not known. Variable structure is a boon for this situation since it can learn and change structure on-line.

Training is used to make the wavelet coefficients better represent the data. It uses the A values and the error generated from stage two to determine which wavelets need adjusting and by how much. Then new coefficients can be created from this and the wavelets updated.

3.1.3.1 Training Wavelets

To train the coefficients of the wavelets that have been hit by the input vector we use Recursive Least Squares (RLS). For the basic algorithm we will perform RLS on every "hit" wavelet simultaneously (in Section 3.3.3 we will see that this is not always necessary).

RLS is performed by keeping global covariance matrices for the input data and the input uncertainty. Encoded in these is all that is necessary to perform incremental coefficient updates (Section 2.3.3 goes more in depth on this subject). The matrices are updated through the use of an information matrix, or A matrix. This matrix is formed by using the A values determined in stage two along with zeros for every wavelet which was not "hit" by the input vector. Any wavelet with a zero in this matrix will not be updated.

With the two covariance matrices, the A matrix and the expected output vector, RLS computes the new coefficient for each wavelet. Each coefficient is assured to be the optimal Least-Squares solution so far (see Section 2.3.3). This coefficient replaces the old coefficient in each of the

wavelet structures, ending stage three. This process is repeated until the stream of input vectors is stopped.

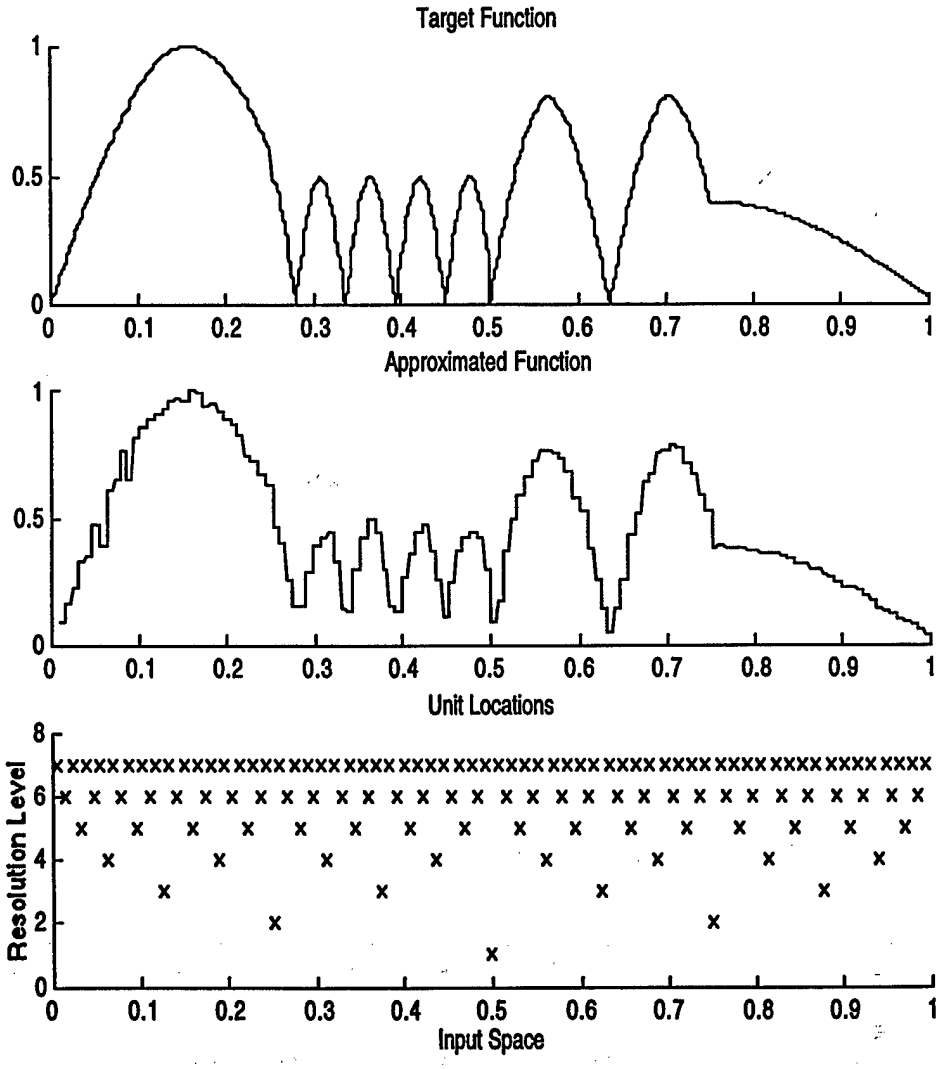


Figure 3.6 Haar Wavelet Approximation a Function

The example presented in Figure 3.6 shows the approximation ability of the basic algorithm. The Haar wavelet was employed and the resolution level was set to 7, giving us 127 wavelet basis units. The data stream consisted

of 3000 random vectors from the input function. The network output nicely captures all of the features of the input function. For comparison, Figure 3.7 shows the same target function approximated using the Mexican Hat wavelet with the basic algorithm. At resolution level 7, it also captures all of the features of the input function.

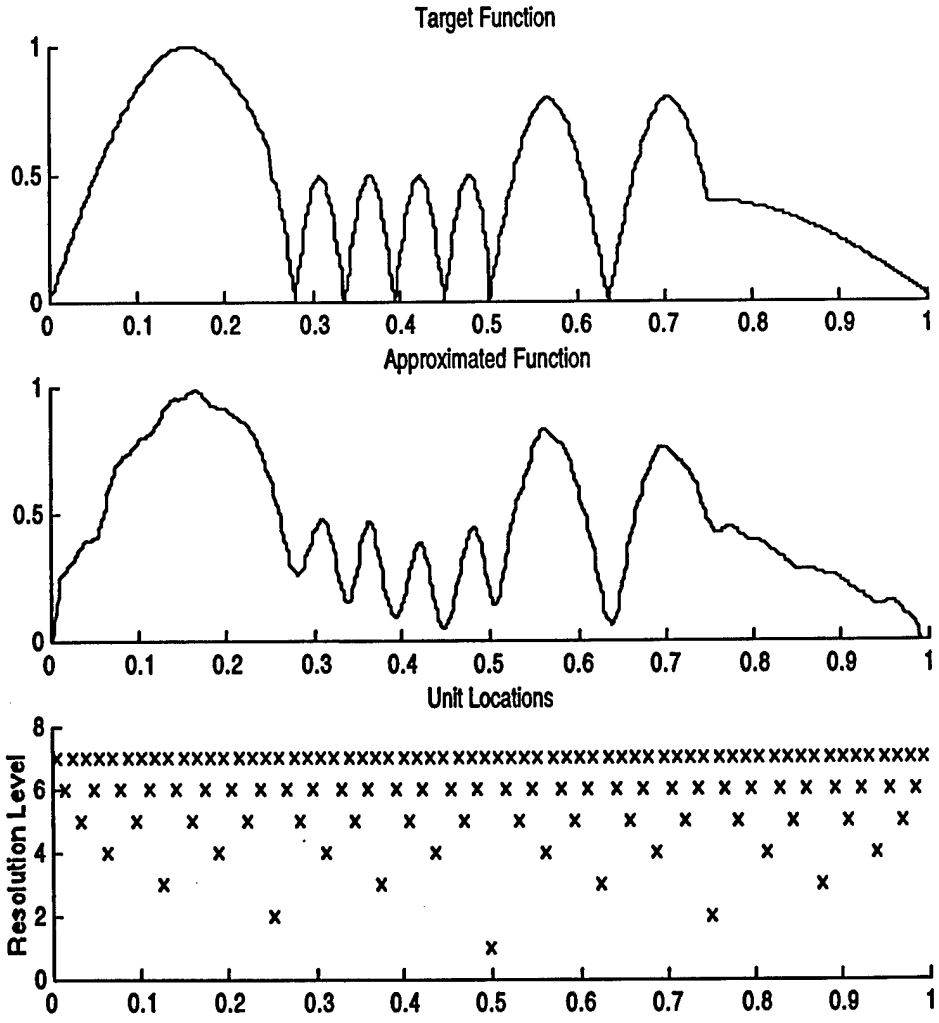


Figure 3.7 Mexican Hat Approximation of a Function

3.1.4 Generalizing the Basic Algorithm

In Sections 3.1.1 through 3.1.3 we presented an algorithm in very basic terms and overlooked a few problems in lieu of a more understandable algorithm. To make the algorithm more general we need to look at the problem of overlapping wavelets and how the scaling function fits into the complete algorithm.

3.1.4.1 Modifications Necessary for Overlapping Wavelets

The use of non-compactly supported wavelets complicates things somewhat. However, this complication only manifests itself in stage one, specifically when we are determining which wavelets are "hit" for an input vector.

Each type of overlapping wavelet has a specific overlapping range. In our multiresolution structure, this value can be represented in terms of how many adjoining wavelets it overlaps into. This number is independent of resolution because the ratio of wavelet size to wavelet space remains the same in our structure. Thus if the overlap value is 2, each wavelet will potentially overlap two wavelets on each side (by convention). We say potentially because the normalization boundaries still stand. Only those wavelets that exist within these boundaries can be overlapped upon. Since overlapping wavelets can extend indefinitely, we need to relax our previous restrictions somewhat. Now, the largest wavelet allowed will be chosen so only a small part

of the function (preferably just the tails) will extend beyond the normalized range. By nature of the location grid, the wavelets above the first will have even less of their functions truncated. The truncated amount will cause a small degradation in the approximation ability, but it will be trivial. Figure 3.8 shows the wavelet hits using overlapping wavelets on the same input as in Figure 3.4.

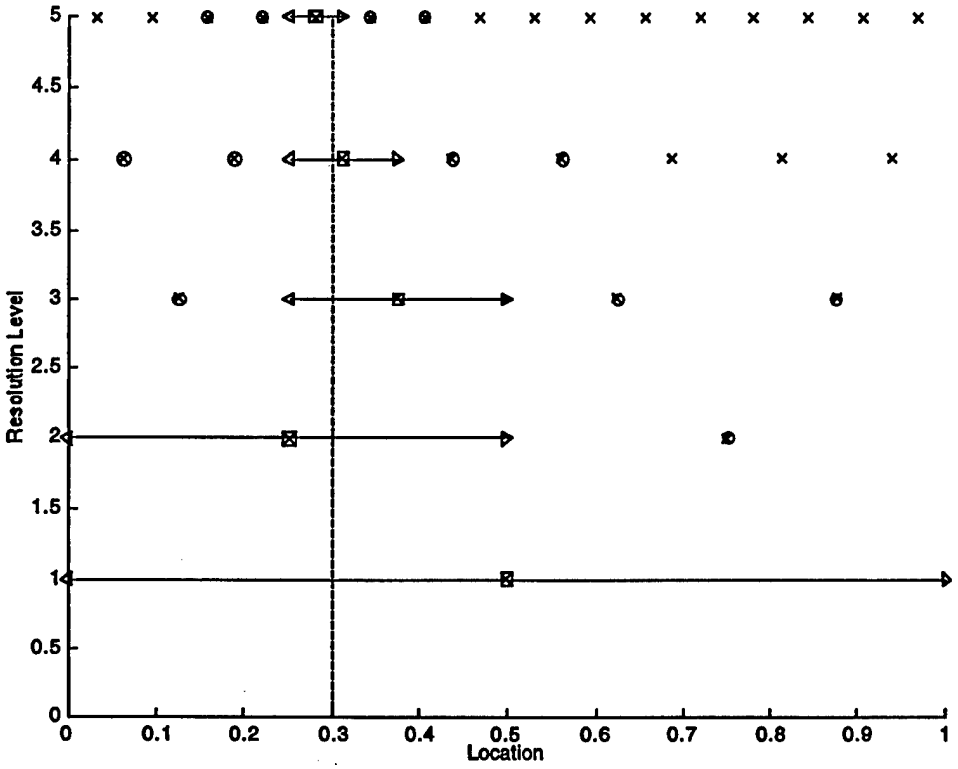


Figure 3.8 Wavelets Hit With Input = .3 and Overlap = 2

Using the overlap value we can find the hits on overlapping wavelets by using Equation 3.1. For each resolution level this will give us the center unit for the input (represented by the squares on Figure 3.8). From this

value we simply include the number of wavelets on each side of the center unit equal to the overlap value (represented by the circled units on Figure 3.8). Thus with n resolution levels the number of wavelet hits will be:

$$n \leq \#\ of\ wavelets\ hit \leq n * (overlap + 1) \quad (3.4)$$

As in Section 3.1.2.2, as the "hit" wavelets are found, they are stored by their identification number to be updated later.

3.1.4.2 Updating the Scaling Coefficient

The scaling function is a special case of a wavelet function, determined by the orthonormal basis chosen (see Section 2.2). Typically, the scaling function is the first thing to be trained and updated since it provides the bias for the approximation.

The same process is used to train the scaling function as we use to train the wavelets. However, the process is quite a bit easier. Since it spans the entire normalized space, it is hit with every input. Additionally, there is only one scaling function, simplifying RLS training in the sense that the P, V, and A matrices are all single values (reference Equation 2.17).

Knowing the above information, stage one is unnecessary. Stage two is completed with the equations:

$$y(\underline{x}) = c \cdot a(\underline{x}) \quad (3.5)$$

$$e = (y^*(\underline{x}) - y(\underline{x}))^2 \quad (3.6)$$

where

$y(\underline{x})$ is the scaling approximation

$y^*(\underline{x})$ is the target vector

e is the scaling error

c is the scaling coefficient

$a(x)$ is the value given by evaluating the scaling function at the input vector

Stage three is completed in the same manner as with the wavelets: the scaling coefficient is determined using the A, P, and V values along with the scaling error.

At this point, something different occurs. Each unit, whether it be a scaling unit or a wavelet unit, takes up a certain portion of the approximation error (we update the units to reduce this error). Since we update the scaling function first and independently of the wavelets (which we can do because of the orthonormal basis), the new error that the wavelets train on is just the scaling error:

$$e' = y^*(\underline{x}) - c' \cdot a(\underline{x}) \qquad (3.7)$$

where

e' is the new error

c' is the new scaling coefficient

This is basically saying that the approximation error for that input is reduced by what the scaling unit is now

evaluated at. The rest of the error which the scaling function cannot absorb must now be taken care of by the wavelets. The scaling function is put into the final algorithm in Section 3.1.5.

3.1.5 Summary of the Basic Algorithm

The basic algorithm is performed one input at a time. For compactly supported wavelets, the algorithm looks like:

- P, V ← wavelet covariance matrices

For some input--

- update the scaling function using RLS
- update the wavelet functions using RLS:
  - A ← empty vector of size = total number of wavelets
  - *for every resolution level--*
    - num ← ⌈input / wavelet_support(i)⌉
    - ID ← get_wave_ID(num, i)
    - A(ID) ← find_Avalue(input, ID)
    - P,V, coefficients ← perform_RLS(A, P, V)
  - wavelets ← new coefficients

Non-compactly supported wavelets require the overlapping wavelets to also be trained.

3.2 Simple Variable Structure

In Section 3.1 we built an algorithm for approximating a function in an on-line manner using a set of known wavelet basis functions. The goal of this section is to take the previous algorithm and modify it to allow wavelets to be added only when they are needed for the approximation. This algorithm will be the basis for the rest of this thesis. To describe the modifications necessary for variable structure we will rely upon the flow chart in Figure 3.1.

In stage one we add more information to be kept by each wavelet. The algorithm for finding "hit" wavelets is also modified to work for unlimited resolution levels and non-existent wavelets (see Section 3.2.1). The additions to stage two deal with computing the new metrics of wavelet hits and local error for each "hit" wavelet (see Section 3.2.2). Finally, stage three's implementation of RLS must be modified to work with a variable number of wavelets and a structure must be created to use the information from stage two to decide when to add new wavelets. New wavelets will be added according to the number of hits and their local error (Section 3.2.3).

3.2.1 Modifications to Stage One

Stage one is really independent of the choice of variable structure. We will still use data normalization and multiresolution, including the location grid shown in Figure

3.3. The only difference is that while the locations of every possible wavelet are known, the wavelets are no longer guaranteed to be there (i.e. they have not been added to the basis). Additionally, some extra information must be kept for each wavelet.

3.2.1.1 The New Wavelet Description

There are some new statistics that will be required for each wavelet in this implementation. In order to conduct variable structure we will need to now keep track of the number of data hits and the local approximation error for each wavelet.

Wavelet:

ID Number	Location	Coefficient	Hits	Error

3.2.1.2 Determining Wavelet Data Hits

In the basic algorithm presented in Section 3.1, we knew the number of resolution levels and that there would be a wavelet in every resolution that would be hit by the data. This is no longer the case in a variable structure situation. Now, there is no limit on the number of resolution levels and there is no guarantee that even if a level exists, that there will be a wavelet which has the input vector within its range.

The solution to these problems is quite easy. First, we make a rule stating that every new wavelet must be placed in the grid and above another existing wavelet (see Figure 3.9).

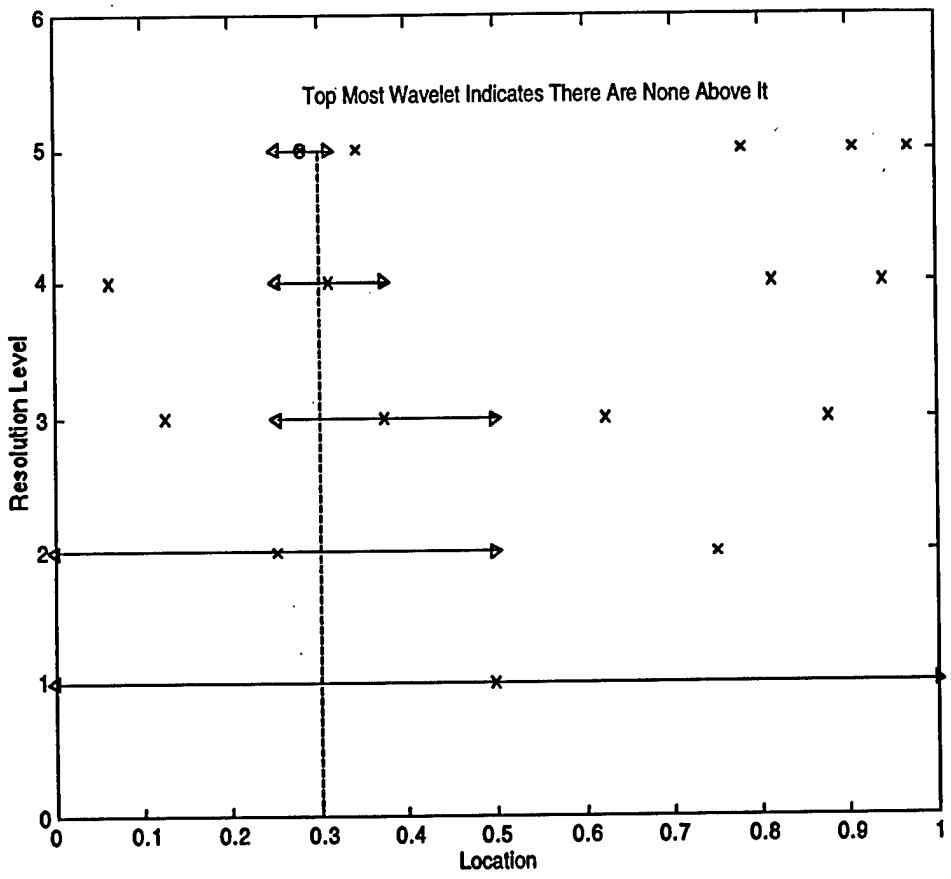


Figure 3.9 Variable Structure Grid Showing Wavelets With Input = .3 in Their Ranges

This sounds restrictive, but it is reasonable. In 3.2.1.1 we saw that we now need to keep statistics on wavelets in order to determine which new wavelets will be added. Every wavelet keeps statistics for the wavelets above it in the next resolution level. This implies that to even consider adding

a new wavelet, the wavelet in the resolution level below it must already be in the basis. We are restricting the power of the representation, but it is necessary if we are going to have the information to determine which wavelets should be added to the basis.

With this new rule, determining which wavelets should be trained on the input data is easy.

- Resolution ← 1
- repeat until wavelet does not exist
    - wavelet = ⌈input / wavelet_range(Resolution)⌉
    - IF wavelet exists
        - store ID to train later
    - resolution += 1

The above pseudo-code relies on a simple rule. A resolution level which does not have a wavelet hit by the input data, implies that there will no more wavelets since they would have to be built above the nonexistent wavelet. This makes determining which wavelets should be trained on a given input as fast as in the basic algorithm in Section 3.1.

3.2.2 Modifications to Stage Two

Previously in stage two, we computed the network output and the approximation error. Now we must also update the hit and error statistics for the highest resolution wavelet with the input data within its range. We only keep statistics for the highest resolution wavelet because it is the only given

wavelet which does not have any wavelets above it (see Figure 3.9). Therefore it is the only wavelet which needs to keep statistics.

Each wavelet must keep statistics for every new wavelet in the next resolution that can be added above it. Using one dimensional input data, this implies that each wavelet must keep statistics for two potential wavelets above it.

The first thing that must be updated is the hits for that particular wavelet.

- *IF hits < min_hits*
    - *hits ← hits + 1*

The variable *min_hits* is the number of hits that are necessary for the wavelet in the next resolution to be added. By requiring a certain number of hits (i.e. 5 hits) we ensure that the errors have had time to stabilize and that we are not adding a new wavelet because of a single spurious input.

We can use the updated hits to update two types of error for each wavelet. The first error is the local average error, or the local L^2 error.

$$avg\_error = avg\_error + (1 / hits) * (approximation\ error - avg\_error)^2 \quad (3.8)$$

where

approximation error is the result of Equation 3.3.

RLS reduces the global average training error so it is necessary to keep the local average error for each wavelet.

This allows us to find the individual wavelets which have large errors. However, it isn't the only error we need to be concerned with.

The second type of error we need to compute is the local maximum error (this is similar to the L^{∞} error). It keeps track of the maximum error experienced by that wavelet so far. The maximum error is used to prevent large error spikes in the approximation, since it is possible for the average error to be low while still having some large errors mixed in. Depending on the application, a new unit might be necessary to bring these large errors down.

3.2.3 Modifications to Stage Three

To accommodate variable structure, the first change needs to be an update to the training algorithm. In Section 3.1 there were a fixed number of units, resulting in a known size for the covariance matrices used by RLS (specifically n x n with n being the number of units). In our variable structure architecture, n increases; implying that our covariance matrices in RLS must also grow. The properties of RLS make this easy. To expand the inverse covariance matrix (P^{-1} - reference Equation 2.17), a row and column is simply concatenated. Thus to add a unit to an m x m inverse covariance matrix, we simply increase the size of the matrix to $(m+1)$ x $(m+1)$, filling the newly created space with zeros. This row/column number now correspond to the new

wavelet and a mapping between them should be kept in order to update coefficients from the RLS algorithm.

This stage is where variable structure actually takes place. With the updated information from stage two, stage three can train, update, and possibly add new wavelets. The initial network has one unit and a scaling function at resolution level one. As inputs are used and the statistics are updated by stage two, units are slowly added to the basis to improve the approximation of the output function.

The decision to add a new unit is based upon three inequalities:

```
hits >= min_hits AND
(avg_error > avg_error_thresh OR
 max_error > max_error_thresh)
```

If this boolean expression is true, the wavelet in the next resolution corresponding to the input data will be added. This means that the topmost wavelet has been hit enough times to give good error information and that at least one of the local approximation errors for that wavelet are larger than the thresholds set by the user for these errors. These thresholds are set according to how much accuracy is required in the approximation. The higher the accuracy, the greater the number of units and time to convergence will be.

After the decision to add a new wavelet has been made, the algorithm repeats. Since wavelets can only be added to the next highest resolution level and only one wavelet will

be hit per level, the maximum number of wavelets added per iteration is one.

3.2.4 Variable Structure Results

One of the most important qualities of a variable structure algorithm is that it only adds the units necessary to approximate the given function. This means that assuming that the function is stationary, at some point the network must reach an equilibrium point where it has enough units to approximate the function. Our algorithm stops adding units once the local errors of all the topmost wavelets are below the given thresholds. Since all of the local errors are below the thresholds, it is easy to see that the global errors must also be below.

Figure 3.10 shows an approximation of an input function using the Haar wavelet and the variable structure algorithm presented in this section. By comparing the locations and frequency of the data (third graph from top) with the input function it is clear that the algorithm uses more wavelets in the high frequency areas of the function and significantly fewer in the low frequency sections. This results in a significant savings of wavelets over the algorithm presented in Section 3.1 (68 vs. 127 wavelets). The wavelet units used are located such that all of the features of the input function are still captured (reference Figure 3.6).

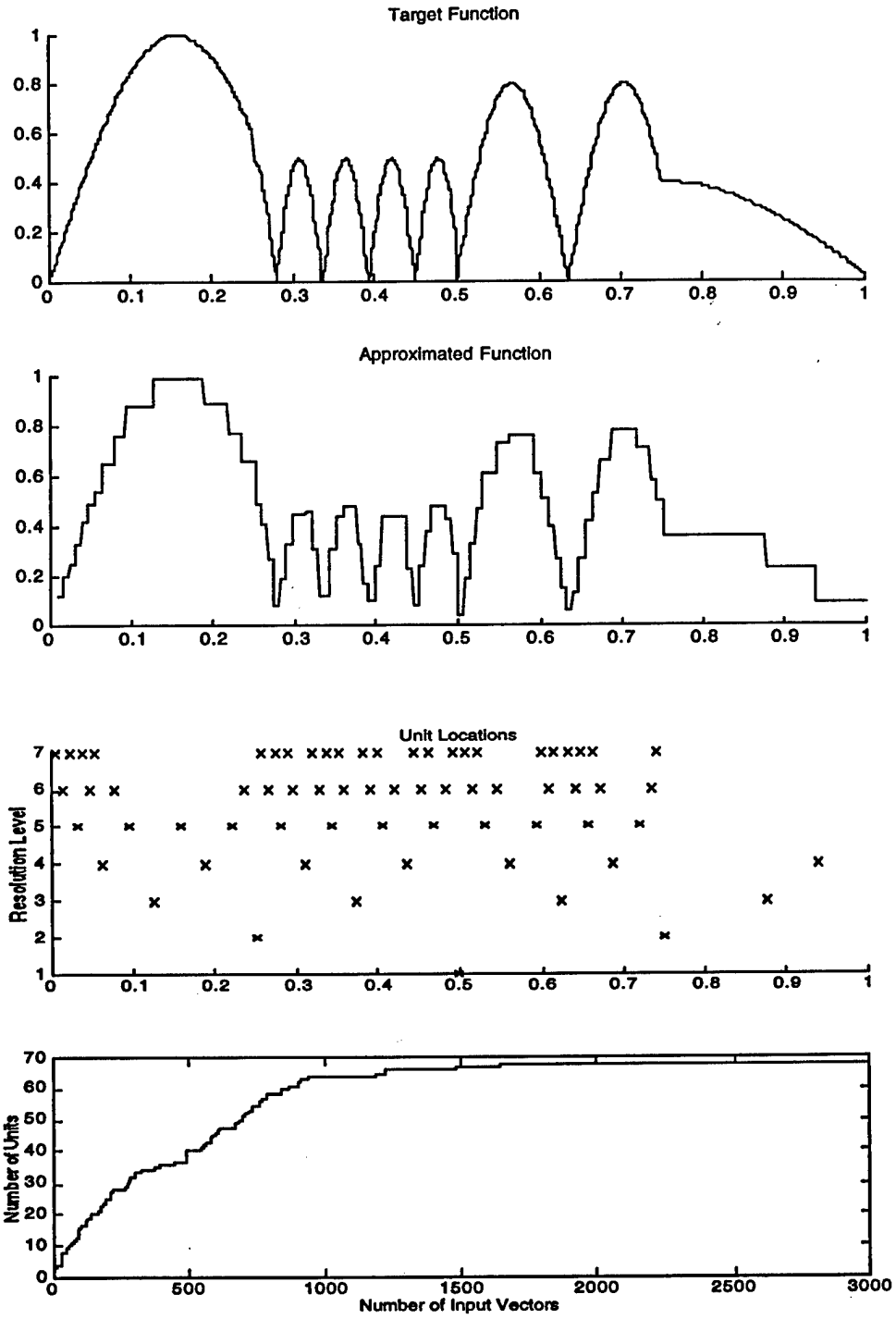


Figure 3.10 Haar Approximation Using Variable Structure

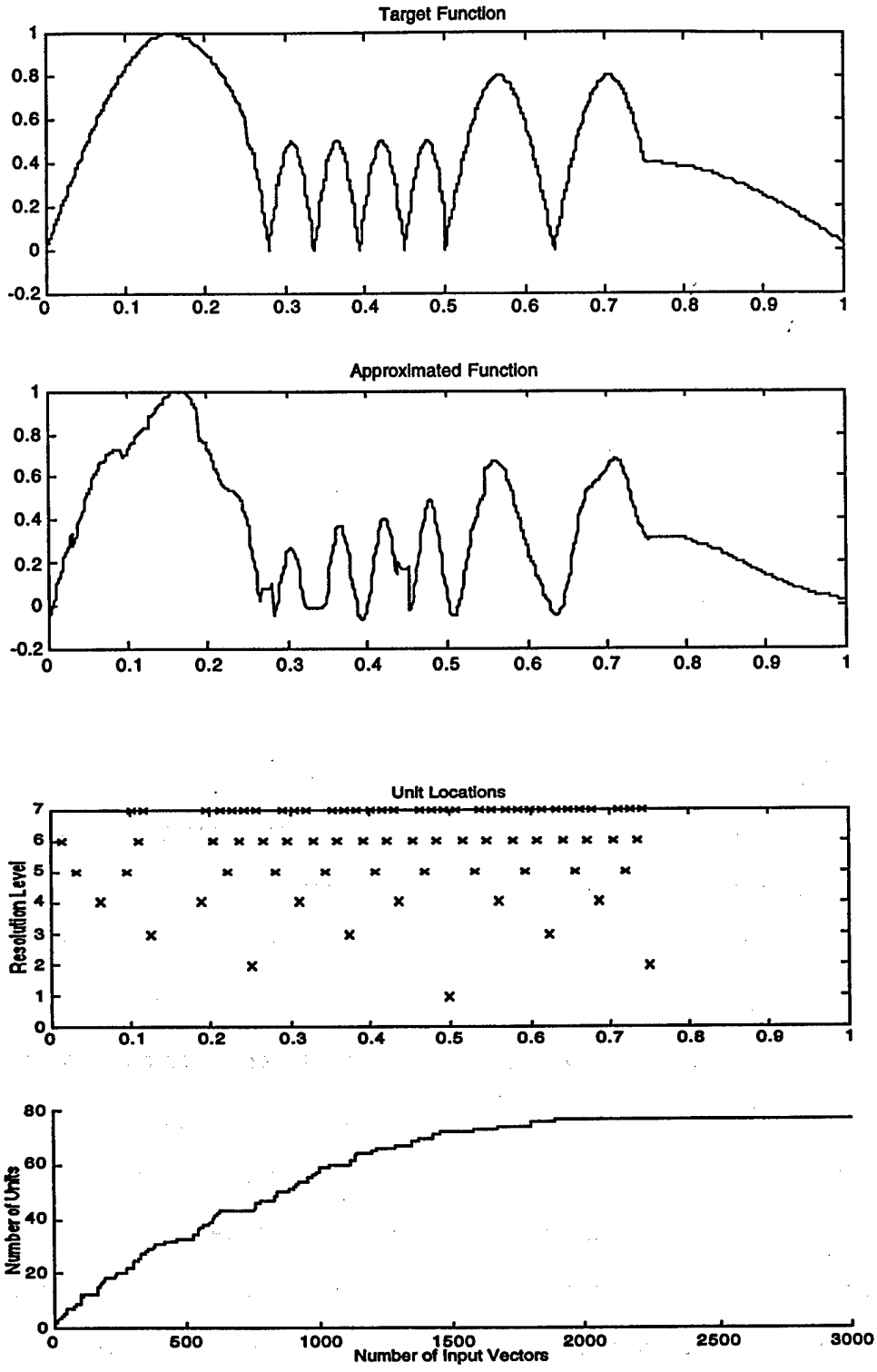


Figure 3.11 Mexican Hat Approximation w/ Variable Structure

Additionally the variable structure algorithm is faster, taking 28.0 seconds to complete while the basic algorithm needed 33.8 seconds. The lowest graph shows the number of wavelets used as a function of number of inputs. It is clear that the addition of wavelets begins to slow around input 1000 and stops altogether near input 1700. At this point the network stops growing. Figure 3.11 shows the same function being approximated using the Mexican Hat wavelet. It requires 79 wavelets to approximate the target function using the same thresholds as in Figures 3.6, 3.7, and 3.10. The resulting function displays the smoothness associated with smooth wavelet approximations. However, the Mexican Hat approximation took significantly more time than the Haar wavelet. The variable structure algorithm took 292.7 seconds versus 340.1 seconds for the basic algorithm. These values are over 10 times larger than the corresponding times for the Haar approximations, with no significant increase in accuracy. Clearly, the Haar wavelet is much more useful in this type of structure.

In this section, we have put together a simple variable structure algorithm which rivals the basic algorithm of Section 3.1. The additional complexity of the implementation is balanced by the large reduction in wavelets and the computations associated with them. Section 3.3 explores additional enhancements that can be made to this algorithm.

3.3 Algorithm Enhancements

3.3.1 Frequency Data

Frequency data is without a doubt the most necessary enhancement to the no-frills variable structure algorithm presented in Section 3.2. When used as part of the criteria for the addition of new wavelets, it prevents the algorithm from overfitting the data, enhancing generalization.

The idea behind frequency data is that the size of a wavelet basis unit should be dependent upon the frequency of the data. Any attempt to use wavelets of higher frequencies than are present in the data will result in basis functions which are unfounded and that can't be properly trained. This is best displayed in the case of the Haar wavelet of Figure 3.12. If wavelet one were to be trained on just the data shown, it may make a very good fit using the left half of the wavelet, but the right half has no information to constrain the approximation. Obviously the wavelet is not the proper size to approximate the data. Wavelet two is a much better choice since both sections of it are constrained by data.

In the case of regularly spaced, one dimensional data, the size of the smallest allowable Haar wavelet is easy to determine:

$$min\ wavelet\ width = 2 \cdot d \quad (3.8)$$

where

d is the distance between neighboring wavelets

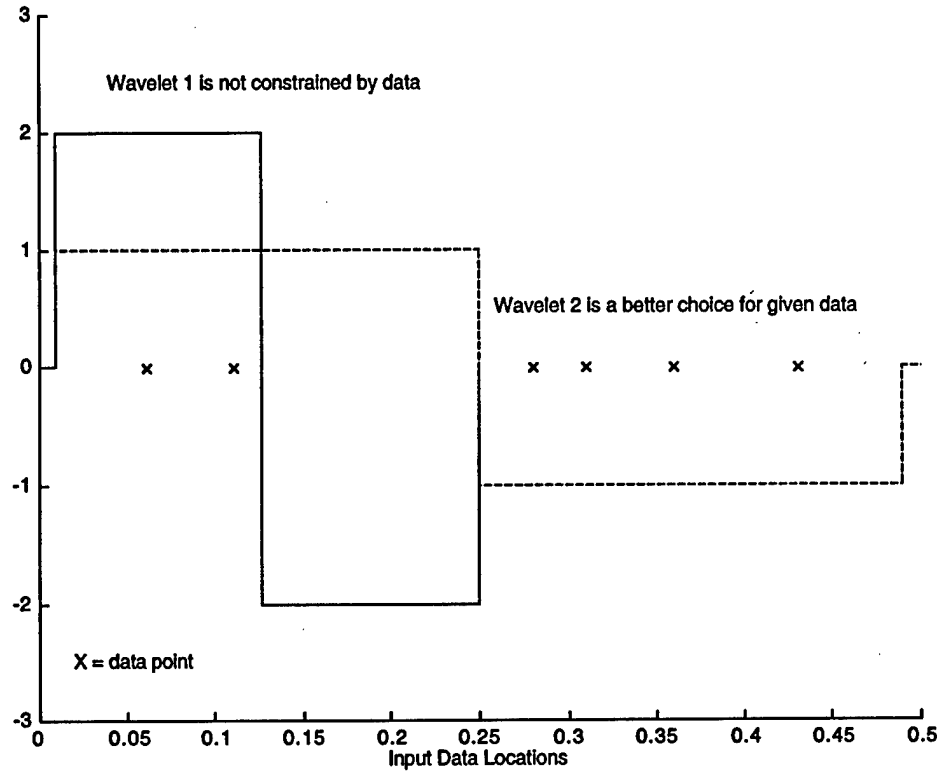


Figure 3.12 Wavelet Constrained by Data

This makes sense because each Haar wavelet will need at least two data points (one on each side) to constrain it (see Figure 3.13). However, our algorithm does not assume regularly spaced data. Luckily, due to our multiresolution structure there is a way to do the same sort of analysis locally.

For one dimensional input, there will be two "bins" that must be filled with data before each wavelet can be added. These bins allow us to keep track of the local data frequency for each wavelet. Multiresolution tells us that for every wavelet used in the one dimensional approximation, there are

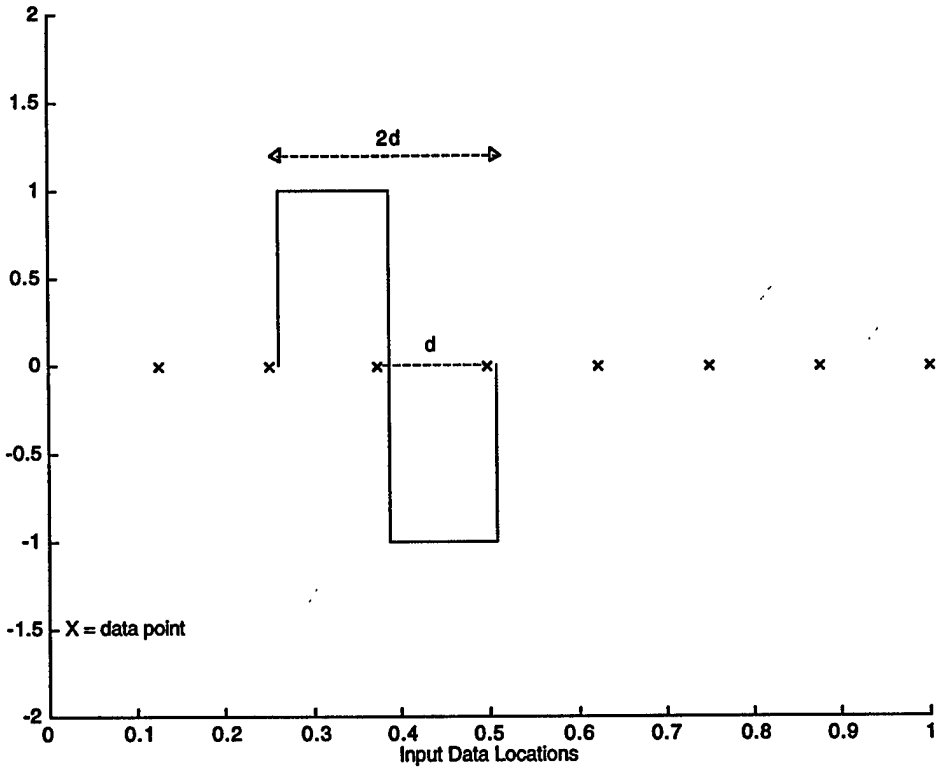


Figure 3.13 Graphical Depiction of Equation 3.8

potentially two wavelets above it in the next resolution. We can use this fact to store the data for the two potential "daughter" wavelets in the current wavelet. This means that in a one dimensional network, each wavelet must keep track of four bins total for the two potential "daughter" wavelets above it (see Figure 3.14).

The requirements for filling the frequency bins can vary. In cases when you wish the network to grow at the fastest possible rate, one data point in each bin will be adequate to constrain the wavelet basis functions (2 data points per new wavelet). In other applications, several

points per bin may be more useful. Our algorithm uses a binary number for frequency with each bin getting one digit (only one data point per bin is necessary). This allows fast retrieval of frequency information by just looking at individual bits and is efficient in the sense that only one variable is used for all of the frequency information.

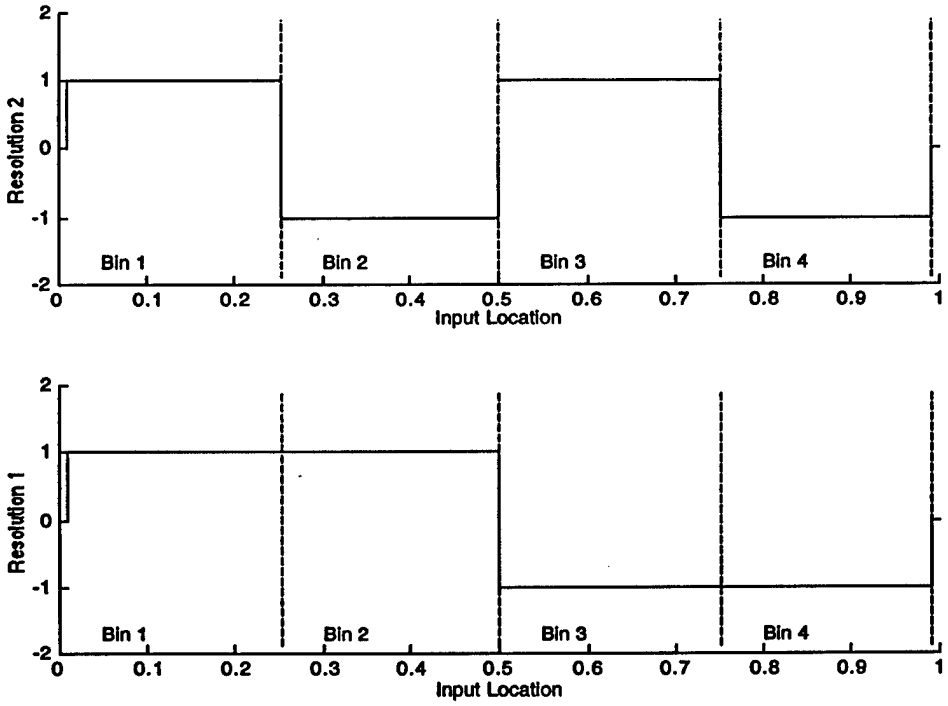


Figure 3.14 The Four Data Frequency Bins for 1-D Data

The approximation of the sparse data (data points represented as X's) in Figure 3.15 illustrate why data frequency is an important criterion for the addition of new wavelets. Instead of just fitting wavelets to the data as in the middle graph, the frequency data only allows wavelets to be placed which are constrained by the input data. Thus in

the third graph we don't see the dramatic spikes at locations .06, .33, and .68. There is a trade off for this better generalization. There are many instances on the graphs were the approximation using frequency data does not approximate the individual points as well. This represents the classic

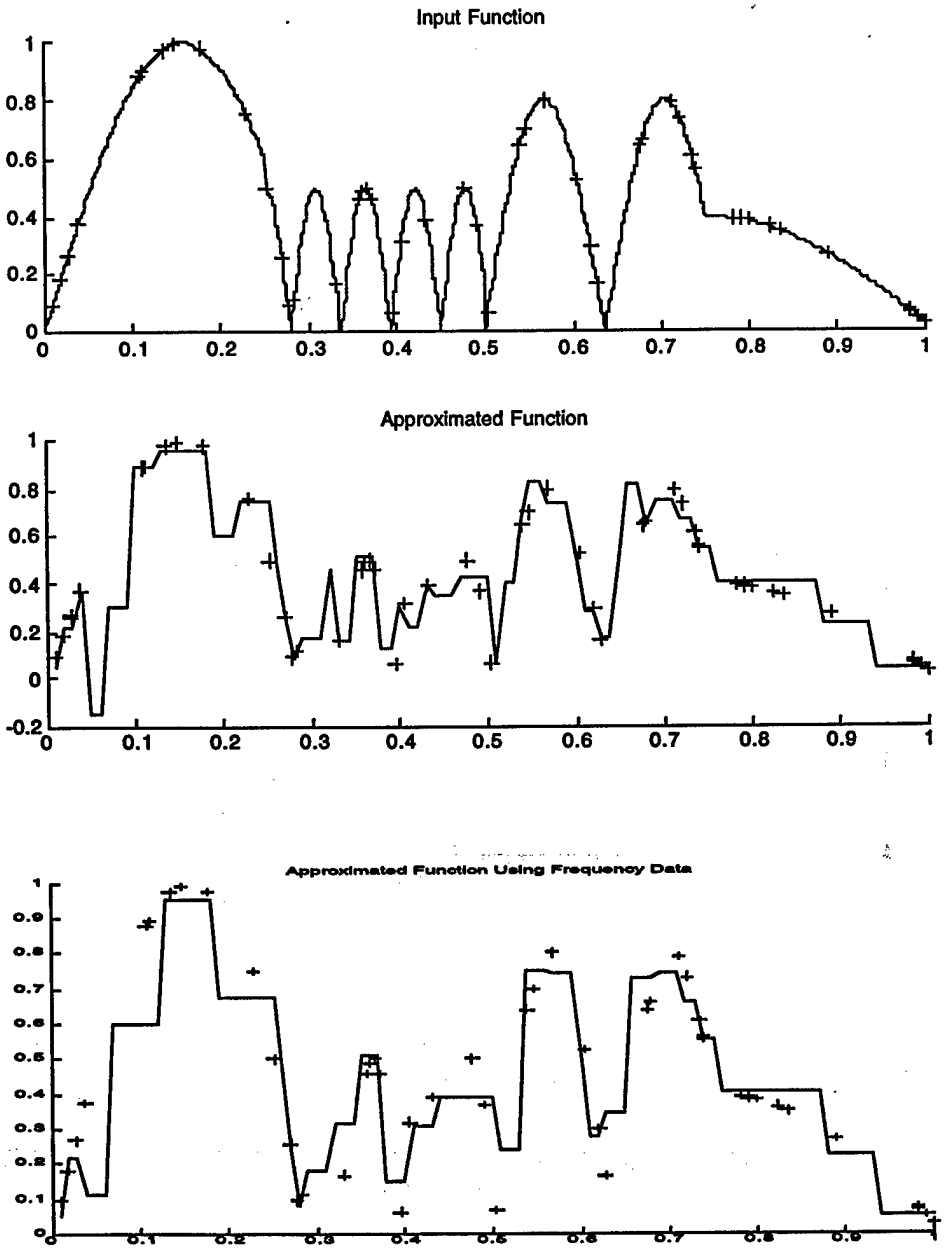


Figure 3.15 Approximation With and Without Frequency Data

trade off between generalization and memorization. In most cases the approximation using data frequency will be more desirable since it is much closer to the "spirit" of the input function.

3.3.2 Inactive Wavelets

The variable structure algorithm outlined in Section 3.2 is somewhat rigid in its implementation. Particularly, the addition of new wavelets is a little too constrained. New wavelets can only be added at the next resolution level, and only above an existing wavelet. What if we have a concentration of data that is at a higher resolution than the wavelets that can currently be added?

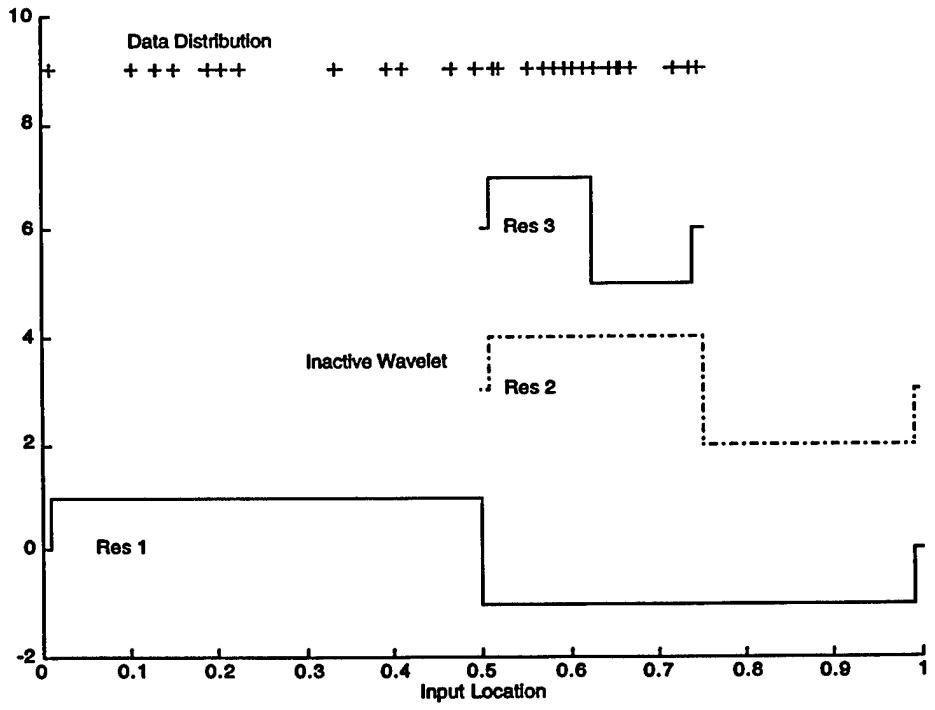


Figure 3.16 Data Distribution Warrants Inactive Wavelet

Approximation error will drop significantly if we are able to bend the rules somewhat and add that small wavelet that the data calls for. In the case of Figure 3.16 the wavelet at resolution level two cannot be added because there is no data present corresponding to its right half (see Section 3.3.1). The wavelet at resolution three, however, has more than enough data present to be added if it is needed by the approximation. Inactive wavelets provide the framework within the variable structure algorithm to allow us to "skip" resolution levels and add higher frequency wavelets to the structure. This in no way changes the viability of the approximation. The orthonormal wavelet basis allows us to use any and all of the wavelets for an approximation. The restriction that wavelets have to be added onto other wavelets is simply to allow us to keep statistics on the error, data frequency, and wavelet hits.

To preserve this system of keeping statistics in the wavelets themselves, place-holder wavelets are added to the structure to hold the position and record statistics. Since these place-holders will not be trained on (since the statistics did not deem them fit to be a real basis unit), we dub them inactive units. These units serve the purpose of being a records keeper so that other wavelets can be added above them. The algorithm for determining if an inactive wavelet should be added is:

- *for each side of the current wavelet (2 sides in a 1-D wavelet)*
    - check wavelet hits >= necessary
    - check wavelet error >= max allowable error
    - check frequency bins (see Section 3.3.1)
        - IF only one of the bins is full THEN
            - Add inactive wavelet

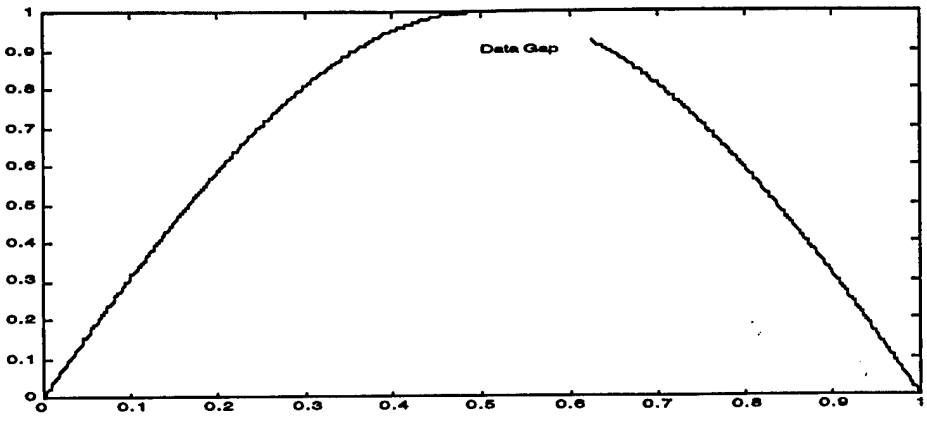
The basic premise of this algorithm is that we are checking to see if only data frequency is stopping a new higher frequency wavelet from being added. We know that the wavelet has been hit by data a significant number of times and that it has been unable to absorb all the error. What we are looking for is if there is a possibility that the next wavelet will never be added because data won't show up in a portion of its range. Notice that we don't automatically add the higher resolution wavelet along with the inactive wavelet. This is because we do not have enough frequency information on the wavelet. We know that it is getting data in its range, but we don't know if both sides of it are getting hit. Therefore we add the inactive to record this information and add the higher resolution wavelet only if it fits all of the requirements. Figure 3.17 shows that there is a significant difference between an approximation with and without inactive wavelets. Figure 3.18 graphs the Haar wavelet units used. The empty circle is the inactive wavelet

placed by the network and the circled units are those that the inactive wavelet allowed. Without the use of inactive wavelets, these wavelets would not be added to the network, giving us the approximation in the middle graph of Figure 3.17. The inactive wavelet allows us to approximate 12.5% more of the input function.

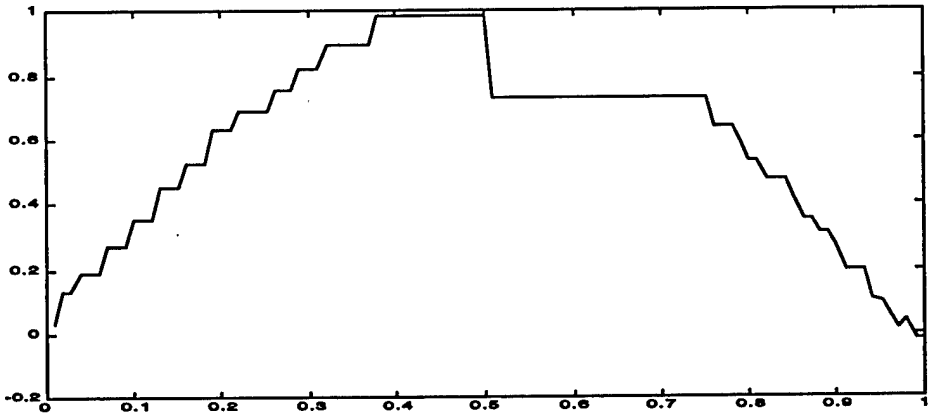
Overall, inactive wavelets are a very useful enhancement to the basic algorithm. They allow the addition of higher frequency wavelets when they are needed, instead of waiting for the entire hierarchy of wavelets to make it up to that level (if they ever do). The only problem is that there is extra overhead required to manage the inactive wavelets and to continue checking to see if they can ever become active (after some more data has been input they may now fulfill the addition requirements). However, this overhead is only necessary once the inactive wavelets have actually been added. No overhead is necessary unless the approximation really needs the higher frequency wavelets.

3.3.3 De-coupled RLS Training

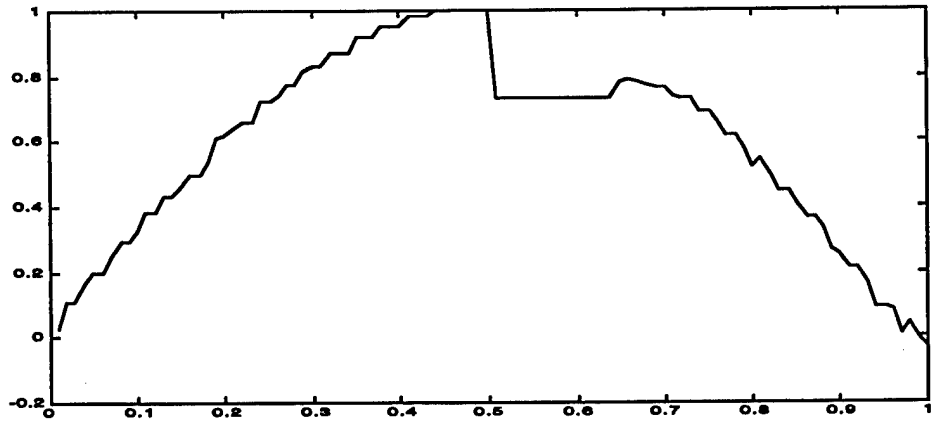
The algorithms presented so far use a simple form of RLS training. For every input a vector is created holding every wavelet's value for that input (a wavelet that is not hit by the data simply has a value of zero). This is then used by RLS to compute the new wavelet coefficients. The problem with this approach is that the covariance matrices used by



Input Function w/ Data Gap



Variable Structure Approximation w/o Inactive Wavelets



Variable Structure Approximation w/ Inactive Wavelets

Figure 3.17 Demonstration of Inactive Wavelets

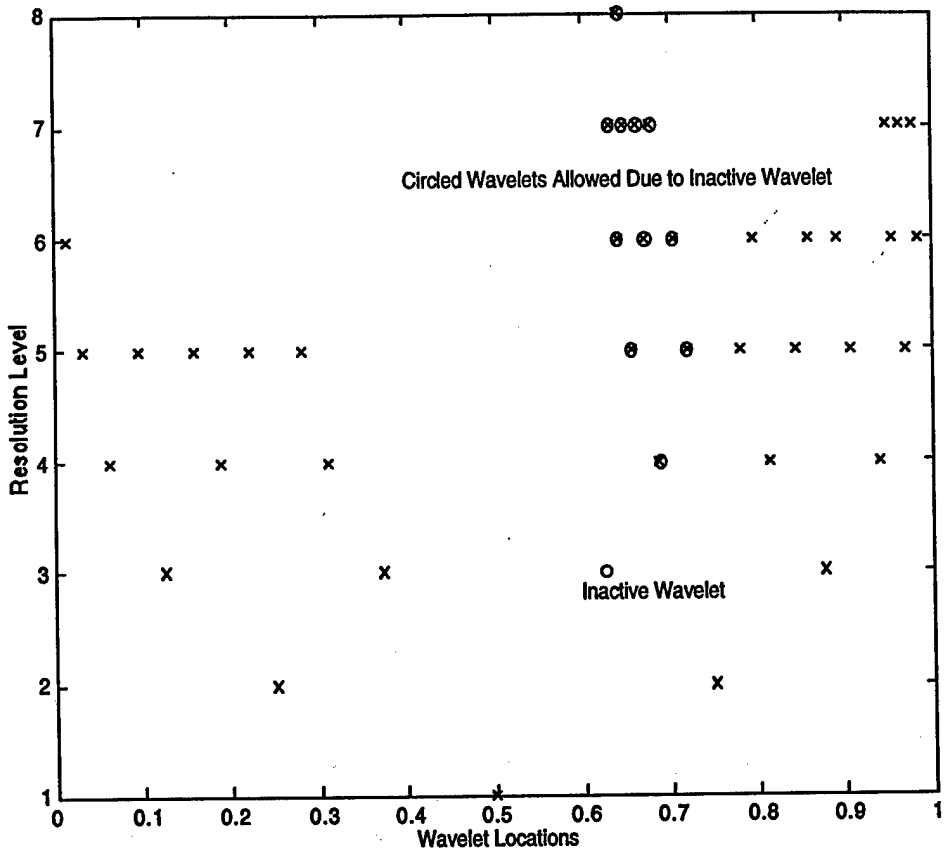


Figure 3.18 Wavelet Unit Locations for Figure 3.15

RLS are *n* x *n* with *n* being the total number of wavelets in the network. These matrices have the potential to get unwieldy and very inefficient since RLS must invert them. To get around this, we need to exploit the orthogonal characteristics of the wavelets.

The purpose of the RLS covariance matrices are to keep track of the interactions *between* the wavelets. Since we are using Haar wavelets, we know that the wavelets are orthogonal

in the limit as the number of input vectors approaches infinity. Thus the outputs of different wavelets are uncorrelated assuming a large amount of well-distributed data. This essentially means that we can perform RLS on each hit wavelet individually and still get a similar result as if we performed RLS on them all at once. The speed and efficiency we gain by doing this is significant since now the covariance matrices only need to be *1* x *1* (each wavelet keeps its own covariance as opposed to one big global covariance matrix). In terms of space, this means that we need to keep *n* *1* x *1* matrices versus 1 *n* x *n* matrix, or *n* versus n^2 terms. Each wavelet can keep its own covariance value, freeing up the space that the large covariance matrices filled. Computationally, de-coupling is also much more efficient since RLS needs to invert the covariance matrices, which is computationally intensive for large matrices. Figure 3.19 compares an approximation performing RLS training on all the units at once versus only performing RLS on individual units. The de-coupled algorithm had a sum squared error 6.8% larger than the full RLS algorithm. This is due to the wavelets not being 100% orthogonal (the data was not dense and regularly spaced). However, the de-coupled algorithm finished in 31.02 seconds compared to 67.97 seconds, resulting in a 54% time gain. In most cases this time savings outweighs the small increase in error.

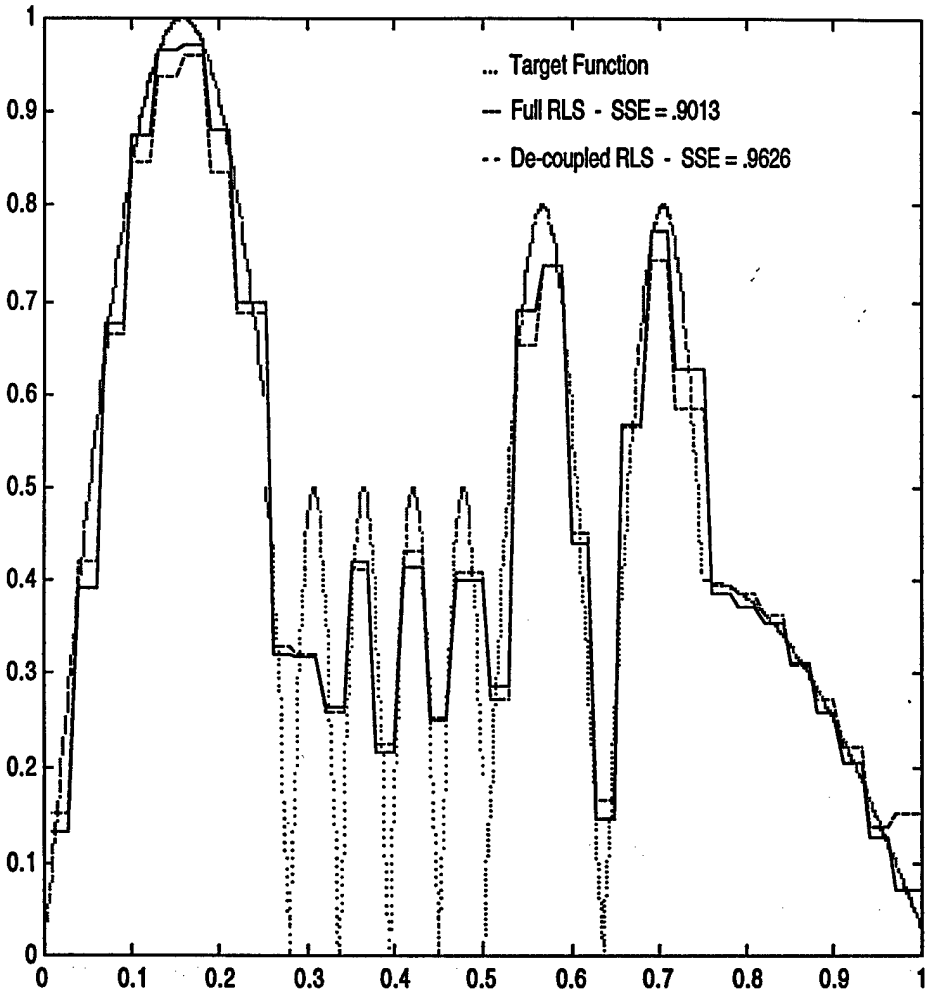


Figure 3.19 Approximation Using Full and De-Coupled RLS Training

The case of non-compactly supported orthogonal wavelets, changes this somewhat. Since the wavelets are still orthogonal, there is covariance between two wavelets at difference resolution levels is still zero. However, non-compactly supported wavelets do interact with each other at

the same resolution level (by overlapping). Thus instead of using RLS to train the units individually, it must train on the subset of wavelets at each resolution level that are hit by the input. This gives us n m x m matrices with $m = 2 \cdot overlap + 1$. In this case, we save space if $m^2 < n$ since we have n m x m matrices as opposed to 1 n x n. Computationally, there is a good chance that de-coupling will be more efficient, even for large m since smaller matrices are much easier to invert than larger ones.

3.3.4 Pruning

In the context of the algorithm presented here, *pruning refers to the process of finding unnecessary or even deleterious bases in the current structure and eliminating them.* It is a relevant topic in any type of approximation where noise or overfitting of the data can be a factor. In the case of a variable structure network employing orthogonal wavelet bases it is very useful since the removal of any of the bases will not affect the others.

There are two problems that pruning can help with: noisy data and overfitting when dealing with areas of the approximation for which there is insufficient data. Noisy data is usually attributed to small, random errors in that data that are often assumed beforehand. It makes it difficult to achieve an accurate approximation because the data itself is not completely accurate. For example, this type of noise can cause small spikes in the network output

due to the wavering nature of the input data. Since the learning algorithm cannot distinguish noisy data from accurate data, these spikes will sometimes be represented using high frequency wavelets. Pruning is very effective in this case because the wavelets tend to be small and conspicuous.

Overfitting in the approximation is usually caused by too little data being used for too much training. In areas where data is scarce, it is often difficult to develop accurate bases. This can cause the algorithm to employ too many wavelet bases without enough data to support them. In our algorithm, this type of error is combated through the use of stored data frequency information and accessed when the addition of new wavelets is necessary. Section 3.3.1 discusses this in more detail. Approximation error can also be due to artifacts left by the bases used in the approximation. An example of this is the square edges of the Haar wavelet left in an approximation of a smooth function. Pruning tends to not be very effective in eliminating this problem.

One benefit of pruning is that there is usually a significant drop in the number of bases necessary to approximate the same function. In an application where space or time to evaluate the bases is limited, pruning can be very helpful. Of course there is always a sacrifice. Although pruning attempts to only remove useless parts of the approximation, it is unlikely that it will do this without

affecting some of the genuine features of the approximated function. However, this tends to give a smoothed, more generalized representation which is desirable in some applications.

Two common ways of implementing pruning are particularly well suited to a wavelet-based variable structure algorithm: cross validation and thresholding methods. *Cross validation attempts to estimate how well the current approximation will fare on unseen data* [13]. It is used to determine if the network is general enough to handle any arbitrary data, or if it has simply memorized the input data. It works by not training the network on some fraction of the known input data. This data is set aside and then used at the end of a training cycle to test the prediction performance of the network on unknown data. It is used to prune units by being performed repeatedly on smaller and smaller incarnations of the same networks, until the best prediction performance is found. Bakshi, Koulouris and Stephanopoulos rely heavily upon this technique in their wavelet network to reduce overfitting and provide generalization [1].

Thresholding methods are used to prune units according to their influence on the approximation as a whole. They basically assume that in any approximation there will be some unimportant details that can be considered to be noise [11]. In terms of our algorithm, this would be decided based on the wavelet coefficients, with the smaller coefficients

considered less influential than the other wavelets. There are many types of thresholding. The two simplest forms are hard and soft thresholding.

Hard thresholding is used when you are looking to reduce the number of units to the minimum necessary. It analyzes the coefficients of every unit and if they fall below a certain magnitude, they are eliminated. Soft thresholding on the other hand, shrinks all coefficients towards the origin [11]. As this happens, the units with smaller coefficients tend to disappear. It is useful for a mild form of smoothing.

Thresholding in general is much faster than cross validation since it can be done in one pass. Additionally, no data needs to be set aside to test the function on. We found thresholding to be useful in reducing the effects of input noise (see Figure 3.20). Hard thresholding is particularly effective at reducing the network to the smallest possible size.

The biggest problem with thresholding is that it introduces a new adjustable parameter into the algorithm. The threshold changes according to the data, so it is difficult to find the correct value. Heuristics are usually employed to determine a reasonable threshold. One such heuristic could be the elimination of a certain percentage of the wavelet units. Depending on the application, the benefits of pruning may be worth the additional trouble of determining a proper threshold.

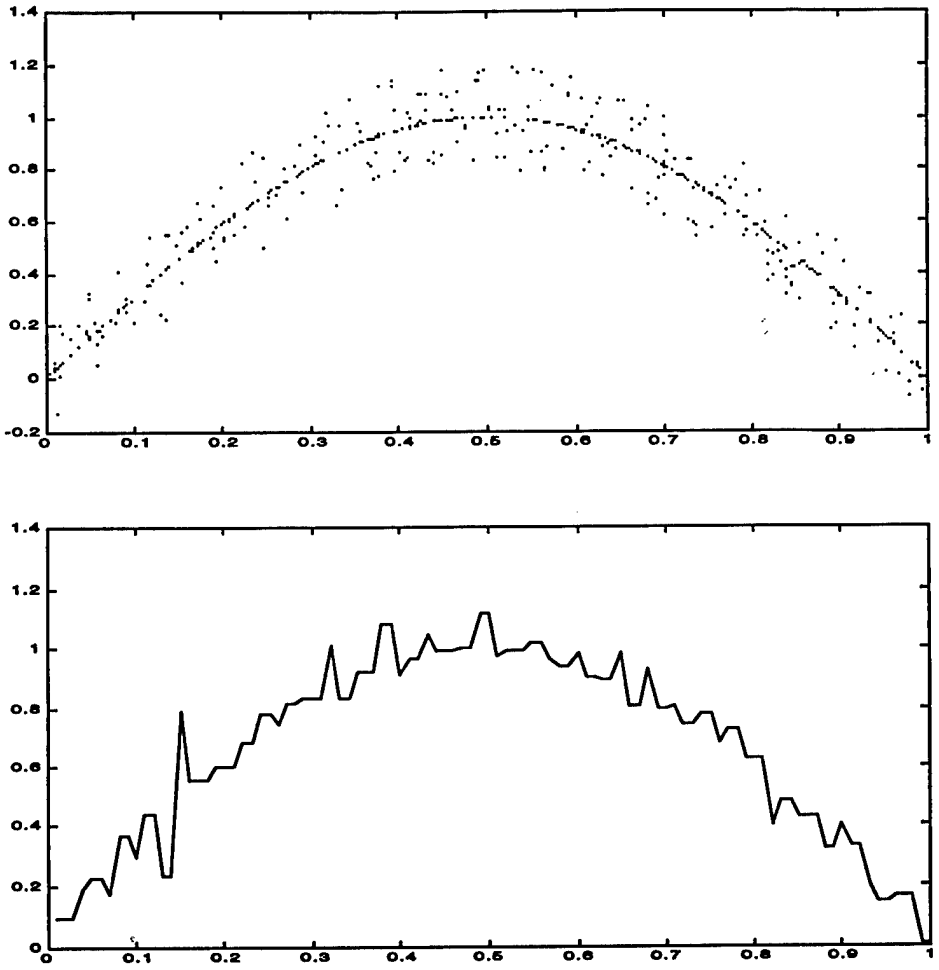


Figure 3.20 Removing Data Noise Using Hard Thresholding

Chapter 4 Multi-Dimensional Networks

In this chapter we will scale up the one dimensional network from Chapter 3 into a general network. This network will be capable of handling n inputs and m outputs with m and n being natural numbers. This change will entail a broadening of some of the concepts discussed in Chapter 3.

4.1 Multiple Inputs

The incorporation of multiple inputs allows us to go beyond the simple one-dimensional approximations and use our network on a much wider array of functions. Accomplishing this will only take a few modifications. To keep continuity, we will organize this section among the same dividing lines as represented in Figure 3.1, discussing only the portions of the previous algorithm which need to be modified for the n-dimensional context.

4.1.1 Determining Hit Wavelets With N-Dimensions

The basis functions that will be used for a n-dimensional network will be n-dimensional themselves. By looking back at Section 2.1.3, we see that these new orthogonal bases are just the tensor products of the one dimensional wavelets presented earlier. As an example, when expanding to two dimensions the resulting equations for the basis will be:

Given $\Psi(x)$, $\Phi(x)$:

	$\Phi(x_2)$	$\Psi(x_2)$
$\Phi(x_1)$	$\Phi(x_1)\Phi(x_2)$	$\Phi(x_1)\Psi(x_2)$
$\Psi(x_1)$	$\Psi(x_1)\Phi(x_2)$	$\Psi(x_1)\Psi(x_2)$

Scaling Function : $\Phi(x_1,x_2) = \Phi(x_1)\Phi(x_2)$

Wavelets:

$$\Psi^1(x_1,x_2) = \Phi(x_1)\Psi(x_2)$$

$$\Psi^2(x_1,x_2) = \Psi(x_1)\Phi(x_2)$$

$$\Psi^3(x_1,x_2) = \Psi(x_1)\Psi(x_2)$$

We can keep the same wavelet description with the minor change of the element *Location* now storing an *n*-vector.

ID Number	Location	Coefficient	Hits	Error

Wavelets are now set in a fixed *n*-dimensional hyper-cube. This hyper-cube is set up in the same manner: the wavelet in resolution one spans the hyper-cube and each resolution above it has wavelets that are half the size of the wavelets from the previous resolution. While we relied upon there only being two potential wavelets at the next highest resolution for each wavelet before, we now have to use the more general rule of 2^n potential wavelets.

Finally, we use the same Equation (3.1):

$$num = \left\lceil input / wavelet\ range \right\rceil$$

to determine which wavelet in a given resolution level has been hit. The only difference is that it must be used for each dimension, or *n* times. This gives us the following algorithm:

- *coordinates* is an empty vector of length *n*
- For *dimension* ← 1:*n*
  - $coordinates(i) \leftarrow \left\lceil \frac{input(i)}{wavelet\_range(i,resolution)} \right\rceil$

where

coordinates is the location of the hit wavelet in the given resolution level

wavelet_range is a value determined by the size of the wavelet

⌈ ⌉ is the ceiling function

4.1.2 Computing Wavelet Statistics With N-Dimensions

The computation of the statistics is the same regardless of the input dimension. However, one thing that does change with dimension is the number of statistics that must be kept. In Section 4.1.1 we determined that each wavelet will have 2^n potential wavelets above it. That means that each wavelet must keep statistics for each of those 2^n wavelets.

One other statistic is altered by multi-dimensional wavelets: frequency data. While this was listed as an enhancement in Chapter 3, it is an important part of our network since it provides the generalization component. The statistics such as hits and errors are the same regardless of the shape of the wavelet. Data frequency information, on the other hand, is directly related to the shape of the wavelet. In the one-dimensional example using the Haar wavelet, there were two regions to keep track of. Now frequency data must be kept for 2^n regions. The implications of this are that as the input dimension increases, so does the amount of data necessary to fulfill the data frequency requirements (since there are more bins to fill). This is a necessary evil in order to ensure that the multi-dimensional wavelets have enough data to support their addition to the network.

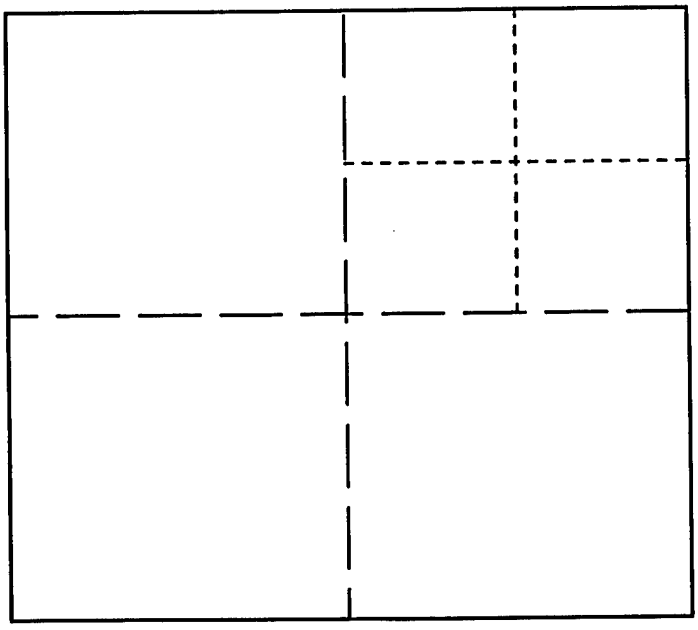


Figure 4.1 Data Frequency Bins for Two Input Dimensions

Figure 4.1 shows the four wavelets at the next higher resolution of the initial resolution 1 wavelet (n = 2 so we have 2^2 wavelets and 2^2 bins per wavelet) with the thick dashed lines. The frequency bins for the wavelet in the upper right are shown with thin dashed lines.

4.1.3 Training N-Dimensional Wavelets

Our new multi-dimensional wavelet bases can be trained much the same way as their one-dimensional counterparts. This is true because we are essentially training on the same thing: the wavelet's value at the input vector vs. the target value. These values are scalar regardless of the input dimension. The only change in the computations is in evaluating the wavelets bases at the input vector.

With the increase in dimension, we must now compute the RLS A matrix a little differently (reference Sections 2.1.2 and 3.1.3.1). In one dimension we had to evaluate each wavelet at the input location. What we are dealing with now is a multi-dimensional wavelet created by the tensor products of one-dimensional wavelets/scaling functions. Thus all we need to do is take the product of each one-dimensional function corresponding to the dimension component of the input. The equation is:

$$a_{d,t}(\underline{x}) = \prod_{j=1}^{n} f_{d,t}(x_j) \qquad (4.1)$$

where

a is the value to be placed in the A matrix or used alone (with de-coupled RLS)

f is the one-dimensional function (wavelet or scaling)

d is the dilation of the function

t is the translation of the function

j is the dimension

x is the input vector

x_j is the input component for dimension j

RLS operates the same as before, taking the results of Equation 4.1 along with the input and target vectors as input and providing the new wavelet coefficients as output.

4.2 Multiple Outputs

While Section 4.1 concentrated on expanding our algorithm to work with multi-dimensional input, a truly general network will also need to be capable of dealing with multiple outputs. Specifically, we want our algorithm to have the ability to approximate a function with m outputs.

Given m outputs, we could break the problem down into m separate functions. Therefore, one solution to this problem is to have m different networks, one for each output. However, every output has the same input vector. We would like to capitalize on this by adjusting our algorithm to accommodate multiple outputs, preventing the overhead and

loss of efficiency associated with multiple copies of the network.

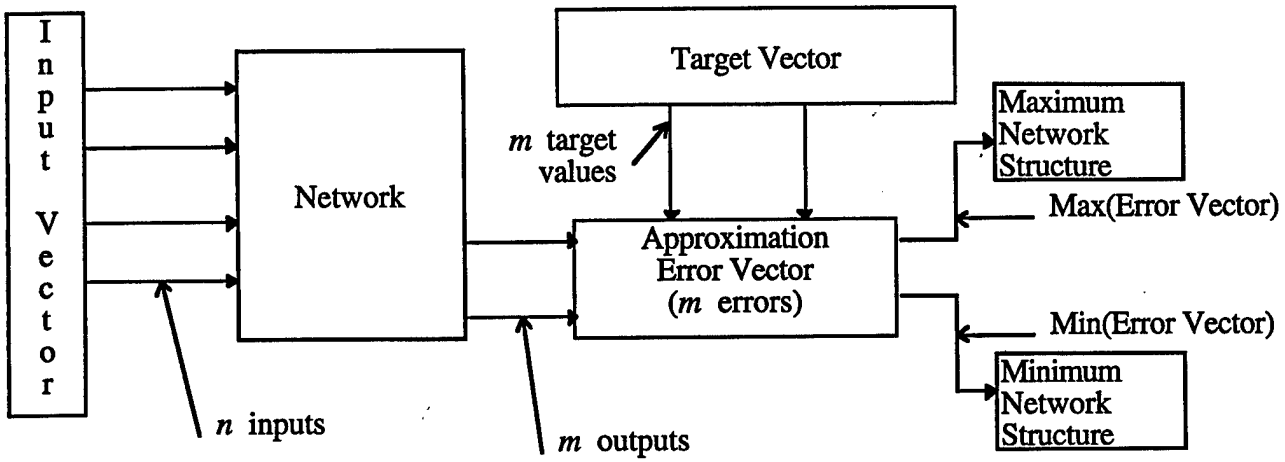


Figure 4.2 Network Flow Chart With N Inputs and M Outputs

Figure 4.2 shows what our new general network looks like. The input vector determines which wavelets are hit by the input and where they are hit (for training and frequency data purposes). The network component takes an input vector in and produces *m* outputs according to the basis functions in the network. The target values can be grouped together into a vector that we will refer to as the target vector. This vector and the network outputs are used to determine how well the network is approximating the target functions, yielding an approximation error vector. Each element in the error vector corresponds to one of the *m* outputs. This error is used to train the wavelet basis units to better resemble the targets. If this were the only use of the network outputs,

then adding multiple outputs would be as simple as just keeping *m* coefficients for each wavelet corresponding to each of the outputs. Unfortunately, the approximation error (which is dependent upon the outputs) is also used as part of the criteria for building the network structure (see Section 3.2.3).

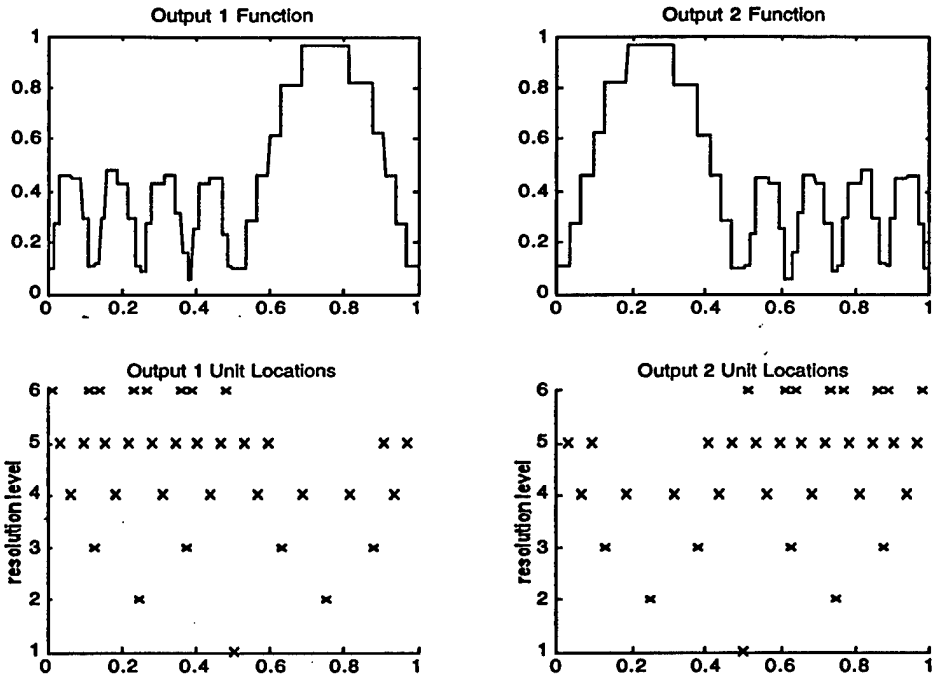
Since the structure of the network is directly related to individual output values, it seems logical that separate networks should be used, one for each output. However, this is very inefficient since the computations associated with finding hit wavelet units and updating their statistics will be repeated *m* times. A method needs to be devised such that the error information can be used to build a network structure that satisfies all of the outputs. We have devised two methods for doing this, the *maximum network*, and the *minimum network*.

4.2.1 The Maximum Network Method

The maximum network method creates the maximum network necessary to satisfy the requirements for *all* of the target values. It is very easy to implement. Each wavelet keeps *m* coefficients corresponding to each output. Network structure is determined by taking the approximation error vector and finding the maximum error. This is the error that is used to determine which new wavelets should be added. This means that wavelets will be added if <u>any</u> of the errors are above the thresholds.

By basing the variable structure component of the network on the maximum error, we do not lose any accuracy on the approximated function. In fact, our accuracy will actually improve since additional wavelets will be used to approximate inputs which would not normally require them. However, our overall efficiency is reduced since these wavelets will be superfluous for most of the output functions. In Figure 4.10 we see an example using two output functions. If we use two separate networks we need to store 35 units and use 43 seconds of computation time for each network (70 units and 86 seconds total). The maximum network combined structure uses 47 units and takes 54 seconds to approximate both outputs, and provides a little better approximation since more wavelets are used.

The maximum network method is very basic and requires no extra statistics, or overhead, to be kept for its use. It is most successful in approximating functions whose output structures do not differ significantly. In these cases the loss of efficiency incurred by training many of the outputs on a larger than necessary network will be lower than training m networks of the various optimal sizes.



Two Outputs and Their Unit Locations

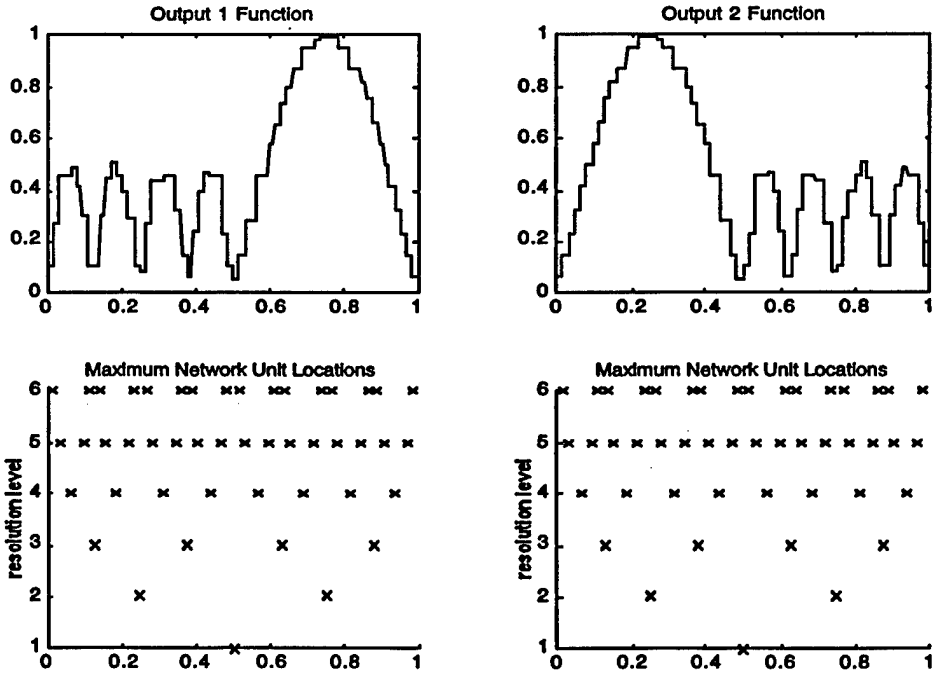


Figure 4.3 Comparison of the Maximum Network Method

4.2.2 The Minimum Network Method

The minimum network method combats the problem of using individual networks by keeping the smallest network structure that is common to all of the target values. This is created by varying the network structure using the smallest of the m approximation errors that are generated by the network. An additional m separate structures must be kept, consisting of the wavelet basis units which must be added to the minimum network in order to approximate the m-*th* output. Since we are dealing with an orthogonal, multiresolution structure, it is no problem to use separate additive structures in this context.

This method gives us the smallest network necessary for all m target values. Additionally, there is no redundant storage of wavelets since the wavelets common to all of the targets are stored in the minimum network. The real drawback to the minimum network method is the increase in overhead necessary to implement it. A mechanism is needed to determine when a wavelet basis function is common to all of the output structures, add it to the common network structure, and remove it from the other structures. When m is large, the loss in efficiency from this mechanism could be prohibitive.

4.3 The Curse of Dimensionality

The most obvious result of allowing multi-dimensional input/output is that the complexity of the algorithm

increases with the dimension. Many of the computations done for a one-dimensional network must now be repeated for each dimension. New network structures must be created to allow multiple outputs. As if this weren't enough, we also have to deal with the added complexity caused by multi-dimensional wavelets. The curse of dimensionality manifests itself with the exponential growth of wavelets as the input dimension is increased.

We mentioned in Section 4.1 that wavelets are built upon one another using the relationship that there are 2^n potential wavelets at the next highest resolution for every wavelet. The multi-dimensional orthogonal wavelets from Chapter 2 require an additional 2^n-1 modes per wavelet. Each mode is itself a wavelet at a different orientation from the other modes. This means that every time we add a wavelet to the network we are actually adding 2^n-1 wavelets each requiring a coefficient to be kept (the other statistics are the same among the modes). It is not difficult to see that in a moderately high input dimension space, our network efficiency is greatly reduced.

Reduced efficiency does not mean that our network is unusable with a large input dimension. On the contrary, it implies that a variable structure algorithm is necessary to approximate in such an environment. With the dramatic increase in the number of potential wavelets at higher resolution levels, it is vital that those levels are only embarked upon when the data absolutely requires it. In the

case of a multi-dimensional function which is complex in all aspects of the function space, no network will have an easy time approximating it.

The problem of 2^n-1 modes per wavelet can also be combated. Certain orientations of the wavelets are going to be much better suited to approximate the target function than others. The orientations which contribute the most to the approximation will have the highest coefficients. Once the different modes have been trained with a certain number of examples, it is then possible to prune the modes which have low coefficients (see Section 3.3.4). If a mode is not contributing much, the overhead required to keep track of it is not necessary. With a good pruning threshold, each wavelet's modes can be kept to the minimum necessary to approximate the function to the desired accuracy. Thus while the curse of dimensionality is daunting, it does not invalidate our algorithm. In fact, we have used multi-dimensional wavelets successfully while still being efficient.

4.4 Some Results Using Multi-Dimensional Inputs

We will use this section to illustrate that our algorithm has essentially remained unchanged; it still performs the same in a higher dimension space. We will show two-dimensional input Figures since these are the most easily viewed. The target function was chosen because it has both low and high frequency components. Additionally, it is a

smooth function, which is more challenging for the piecewise-constant Haar wavelet. The approximations were wholly based upon the error thresholds, meaning that they were allowed to approximate to any resolution in order to achieve errors less than the thresholds. These thresholds are input by the user and were discussed in Section 3.2.3. The *threshold for average error* ensures that the local average error of each wavelet is below a certain level. The *threshold for maximum error* forces each wavelet to have an average local maximum error less than the threshold.

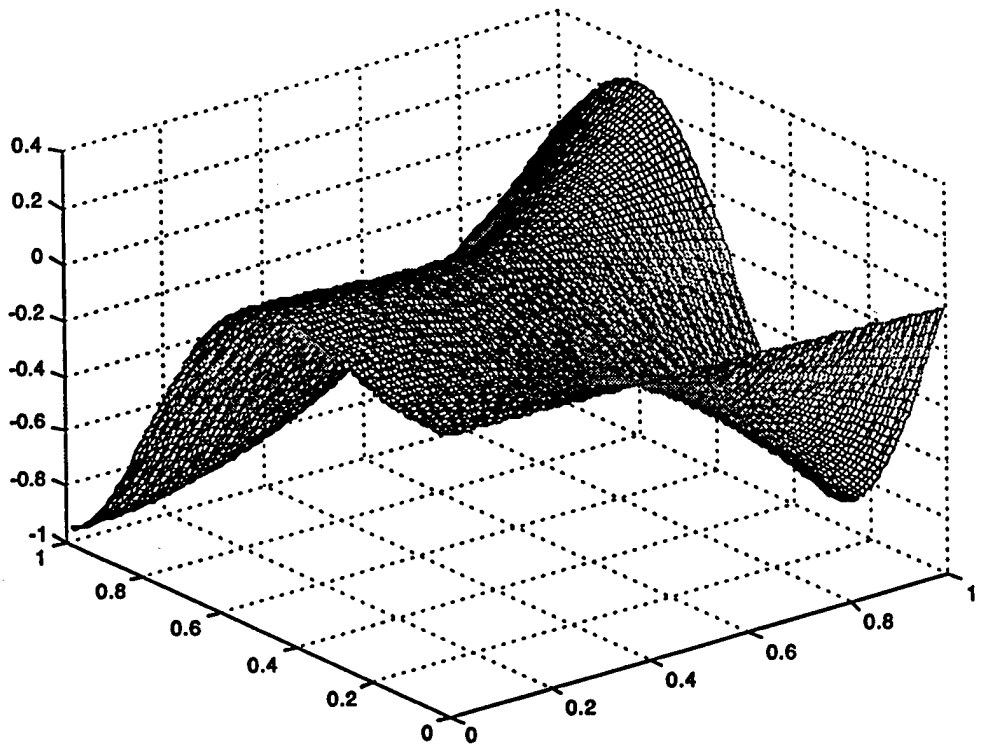


Figure 4.4 The Target Function

Figure 4.4 shows the target function and Figures 4.5 - 4.7 show approximations of this function starting coarse and then adding more and more details. The included tables show the relevant statistics for these approximations

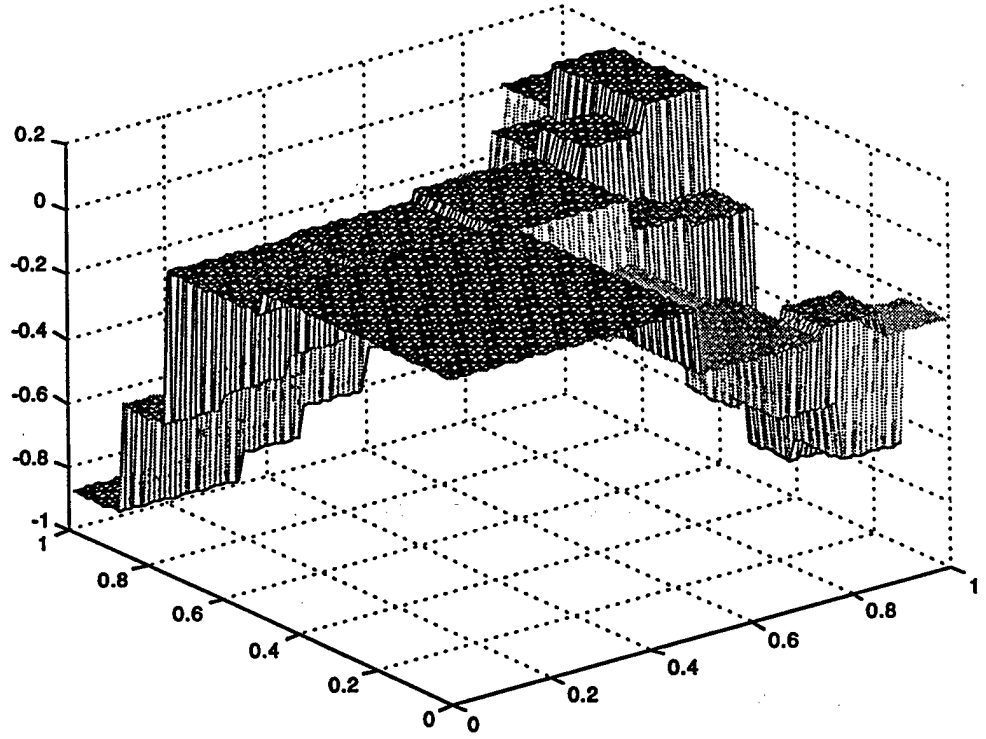


Figure 4.5 Approximation # 1 (coarsest)

Approximation # 1

user input	Threshold for Average Error	.05
	Threshold for Maximum Error	.1
results	Number of Wavelets Used	11
	Computation Time	45 seconds

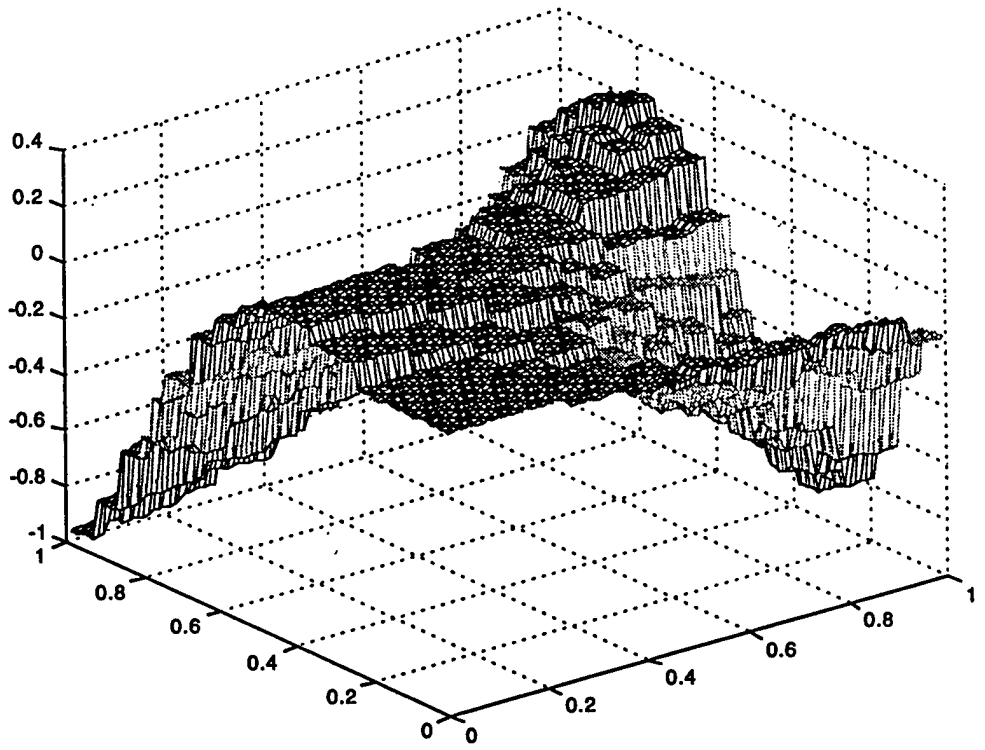


Figure 4.6 Approximation # 2

Approximation # 2

user input	Threshold for Average Error	.005
	Threshold for Maximum Error	.01
results	Number of Wavelets Used	60
	Computation Time	74 seconds

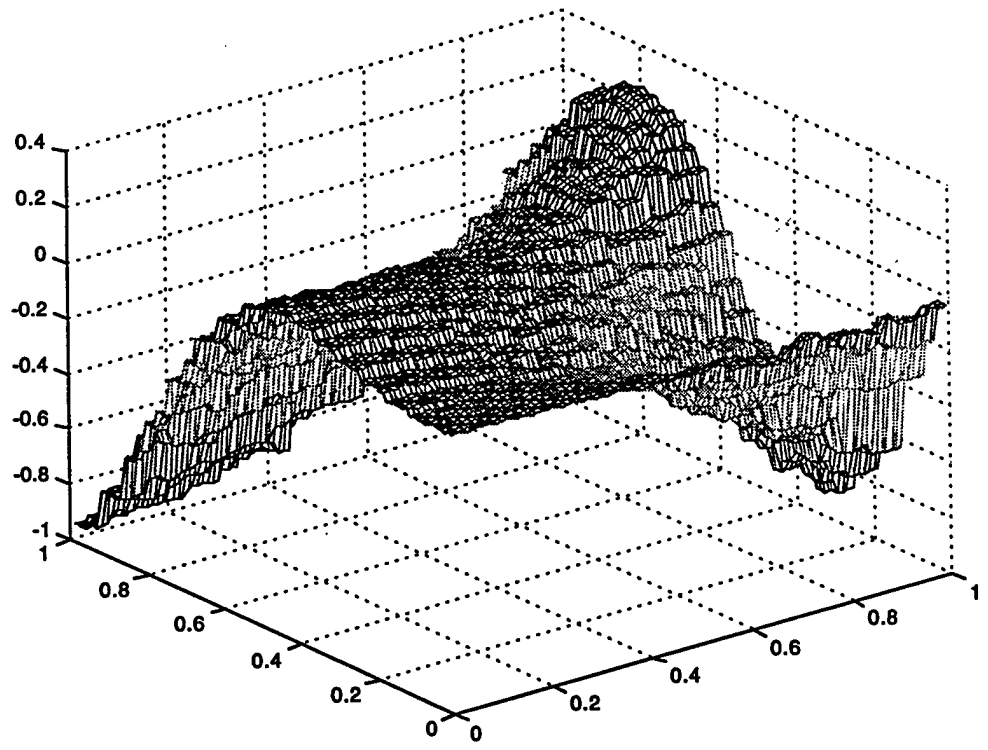


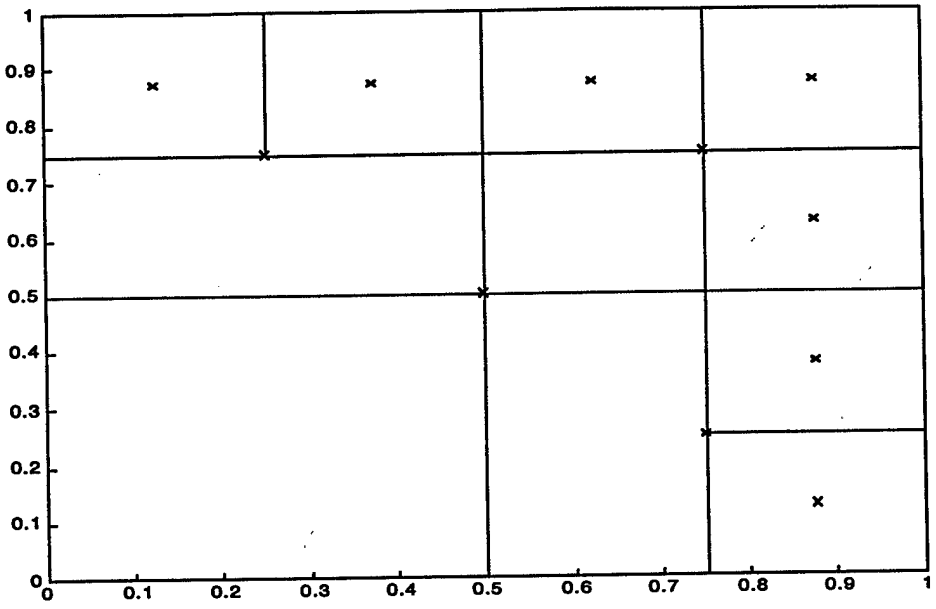
Figure 4.7 Approximation # 3 (finest)

Approximation # 3

user input	Threshold for Average Error	.0005
	Threshold for Maximum Error	.001
results	Number of Wavelets Used	235
	Computation Time	133 seconds

Looking at the three approximations, it is clear that as the error thresholds are reduced, a greater number of higher resolution wavelets are required. Approximation one is very crude and boxy, but it still successfully captures the essential features of the target function. As we reduce the error thresholds, finer details are brought out in the approximation by the higher frequency wavelets. Approximation two reduces the error from approximation one by 73%. Approximation three's error is only 9% of approximation one's error. Figure 4.8 shows that approximation one only uses 11 wavelets while approximation 3 needs 235 wavelets to achieve its desired accuracy. We can see the same type of phenomenon with the one-dimensional data in Figure 2.2. The advantage of our algorithm is that the variable structure set up is the same regardless of the input space dimension. Chapter 5 will show additional results by applying our algorithm to a more realistic problem.

Approximation # 1



Approximation # 3

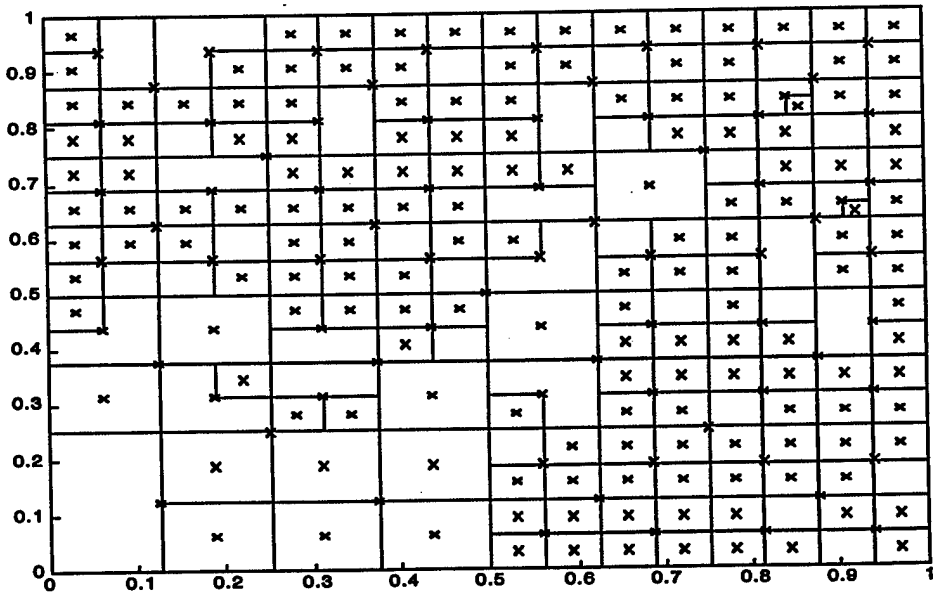


Figure 4.8 Wavelet Coverage for Approximations # 1 and # 3

5 Reinforcement Learning Experiments

The previous chapters have relied upon simple functions and examples to illustrate the different features of our algorithm. In this chapter we will now apply our algorithm to a more difficult and realistic problem and see how it fares with the increased complexity and limitations imposed. Reinforcement learning was chosen because of its interesting problems along with the fact that neural networks are often used as components in these types of problems.

5.1 The Puck on the Hill Problem Description

A well known problem used in reinforcement learning trials is the *puck on the hill* problem.

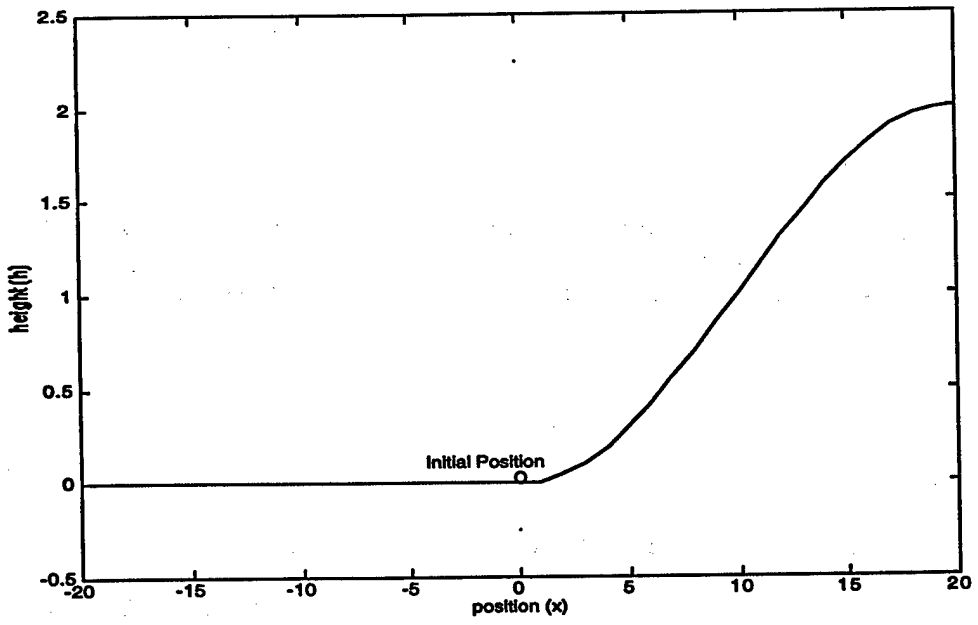


Figure 5.1 Depiction of the Puck on the Hill Problem

Like the title indicates, the problem consists of a puck that must climb a hill (see Figure 5.1). The puck can be pushed either forward or backward. The hill is purposely too steep and the maximum allowable force too small to simply push the puck up the hill. Instead, some momentum must be gained first. The specifics of the problem are:

States:

Position (x) : [-20, 20] meters
Velocity (v) : [-5, 5] meters/second
Control (u) : [-1, 1] newtons

Hill: [x > 0 : 1 - cos(x * π/20)
x <= 0 : 0]

example:

x	h	∂x/∂h
0.0	0.0	0.00
5.0	0.3	0.11
10.0	1.0	0.16
15.0	1.7	0.11
20.0	2.0	0.00

Misc:

Weight of Puck: 1 kg
$g = 9.8\ m/s^2$

When starting at x = 0.0, v = 0.0 and using u = 1.0 (maximum forward force), it is only possible for the puck to move as high as x = 10.5. The solution to the problem is to first move the puck backwards and accelerate along the flat portion of the space before encountering the hill.

5.1.1 Normalizing the Data for Use with Our Network

Before we can apply the problem in Section 5.1, the data must be normalized to work with our network. In Section 3.1.1.2 we discussed the reasoning behind our network needing

normalized data. This in no way changes the problem, it is just a step that formats the data so that the algorithm can rely upon certain boundaries. In our case, this normalization is easy. We already have explicit boundaries for the problem. All we need to do is scale the input data so that it is between 0 and 1 (see Equation 5.1).

$$normdata = \frac{data - \min}{\max - \min} \tag{5.1}$$

where

data is the input to be normalized

min is the smallest allowable input value

max is the largest allowable input value

5.2 Learning the State Transition Function

In reinforcement learning terms, the state transition function is the function which determines what the next state is. Given a state-action pair:

$$\{states\} \times \{actions\} \rightarrow \{states\}$$

The interesting thing about this function is that in reinforcement learning problems it is not known by the learning agent. The learning agent must explore the state-action space to build an internal model for the environment [13]. This idea makes reinforcement learning particularly attractive for problems in which we do not know or cannot easily define the state transition function.

Some algorithms have been proposed to use this state transition model to improve the efficiency of reinforcement

learning [15]. The idea is that if this model can be learned and accessed independently by the learning agent, it can be used to simulate future actions by the agent. These future actions could be considered a sort of implicit planning by the agent to find the best path to the goal state without necessarily visiting every path and performing backups [12]. The result is a much faster and efficient way for the agent to solve the problem.

The puck on the hill problem has a known transition function (Equation 5.2) but for the purposes of representing it as a reinforcement learning problem it is assumed that this function is unknown.

$$\begin{aligned} a[t] &= \frac{u[t]-9.8\cdot slope}{1+slope^2} \\ v[t+1] &= v[t]+a[t]\cdot\Delta t \\ x[t+1] &= x[t]+v[t]\cdot\Delta t \end{aligned} \tag{5.2}$$

where

$u[t]$ is the control at time step t

$a[t]$ is the acceleration of the puck at time step t

$v[t]$ is the velocity of the puck at time step t

$x[t]$ is the position of the puck at time step t

Δt is the size of the time step

For this experiment, we will attempt to learn the transition function for the puck on the hill problem. This will require a network with three inputs (current position, current velocity, and control) and two outputs (new position, new velocity). Figure 5.2 shows a graphical representation of the network.

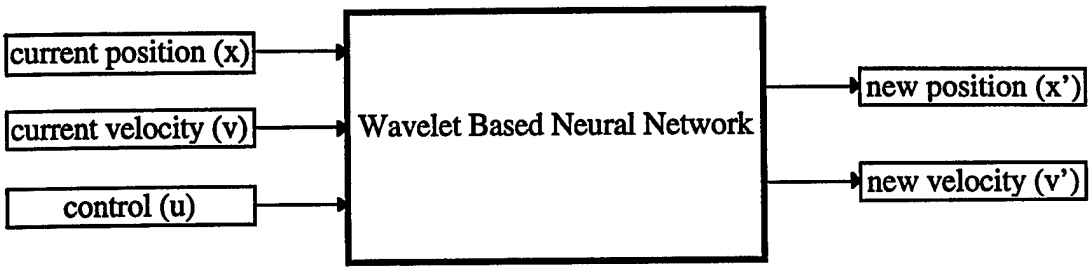


Figure 5.2 State Transition Network Layout

The agent is assumed to have no knowledge of the state transitions and all information is gleaned through exploration along *trajectories*. Each trajectory begins at a random state and a specified number of random actions are generated. For each action along this trajectory, the network is queried, giving its estimate for the next state and then trained using the state that the agent actually transitions to. After a specified number of trajectories, the network is tested to see if it can approximate the transitions necessary to solve the puck on the hill problem.

The state transition problem was set up using random initial positions and trajectories of length 20 (the network trains on 20 actions from the initial position). At every 1000 epochs we used a simulation to test performance. The simulation used known trajectories of length 15. The network began in the start state of each trajectory and used the same actions. The end state of the model was then compared to the end state of the actual system to determine if the simulation was a success. Since continuous data was used, an error margin of ten percent was used to make the problem more

feasible. To pass the simulation, ten trajectories had to be successfully negotiated within the error margin.

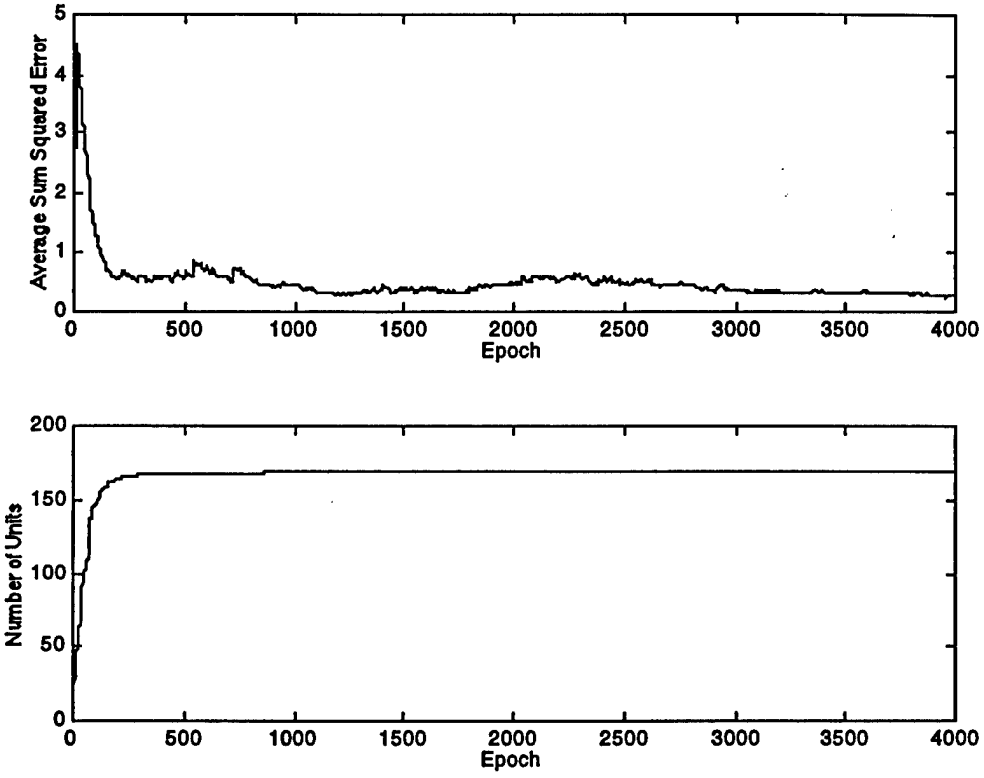


Figure 5.3 Learning Results for State Transition Function

The network performed well on the state transition function trial. As seen in Figure 5.3, training completed after 4000 epochs. At completion, the network was able to successfully follow all ten simulated trajectories. The error plot in Figure 5.3 shows the sharp spike in error while the network is still too small to learn the transition functions. As units are added, the error decreases rapidly, ending with a long period of fine tuning as the network learns the additional information. It is interesting to note

that after approximately 900 trajectories, the network added all of the units necessary for the approximation. While the results of this experiment were good, the time taken to complete training was not. The network needed *16.4 hours* to train on 4000 trajectories (approx. 1.4 data vectors per second). The dimension of the problem (3 input and 2 output dimensions) caused this poor time performance. Chapter 4.3 discusses the curse of dimensionality associated with this algorithm and this test unfortunately supports those warnings. In the recommendations section of Chapter 6 we will talk about how this problem can be combated.

5.3 Learning the Puck on the Hill Problem

Section 5.2 allowed us to test our algorithm on an application useful in the realm of reinforcement learning. However, the problem was basically a multi-dimensional, supervised learning problem. It really doesn't allow us to see how applicable the algorithm in this thesis is to a true reinforcement learning problem. To remedy this, we will use our algorithm to learn the puck on the hill problem. Q-learning will be used for this example. Russell and Norvig provide an excellent tutorial on reinforcement learning and Q-learning [13].

One of the interesting differences of reinforcement learning problems is that they require a network to be tolerant of non-stationary data. Non-stationary data is just that, it does not always give the same output for the same

input. Often times this is associated with some type of error in the measurement, but in this case it is caused by the fact that the network has very little information initially. The *Goal_State* contains all of the information and the other states are defined by their relative distance from it. Until the *Goal_State* is actually visited and enough epochs have passed to allow that information to propagate backward, any query to the network will result in bad information. Thus the longer the algorithm trains, the better the information should get.

To allow for non-stationary data, we employed the age-weighted RLS training algorithm from Section 2.3. Equation 2.20 restated is:

$$P_i^{-1} = \lambda P_{i-1}^{-1} + A_i^T A_i \tag{5.4}$$

From this, we can see that the inverse covariance matrix (P^{-1}) is discounted at each iteration, weighting the newest data the highest. Before diving right into the puck on the hill problem, we tested the algorithm on some simple training sets simulating non-stationary data.

5.3.1 Testing the Network with Non-stationary Data

Before testing on a full-fledged reinforcement learning problem we first tested the network's capabilities with some simple test sets of non-stationary data. We created test sets that start out with very bad data and then progressively get better, ending with accurate data. There are other ways to perform this type of test, but this method closely

simulates the nature of reinforcement learning, with very bad values in the beginning and better values as time goes by. What we are looking for is the network to remain stable and be able to adjust its parameters to compensate for the fluctuating data. Also, given enough good data, we want the approximation to eventually converge to the target function.

The first example set was created by adding a constant value to the target function, $y(x) = sin(2\pi x)$:

Data Points 1-1000 : $\sin(2\pi x) + .5$
Data Points 1001-2000 : $\sin(2\pi x) + .25$
Data Points 2001- ... : $\sin(2\pi x)$ (the target function)

We will call this the *constant value test*. Figure 5.3 shows this graphically.

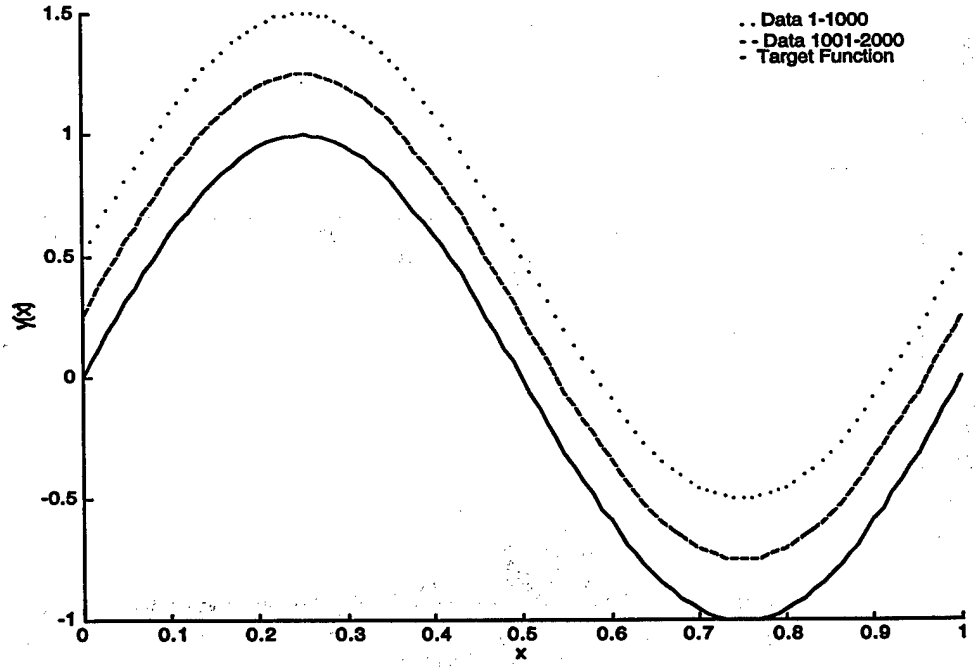


Figure 5.3 Non-stationary Data Used for Constant Value Test

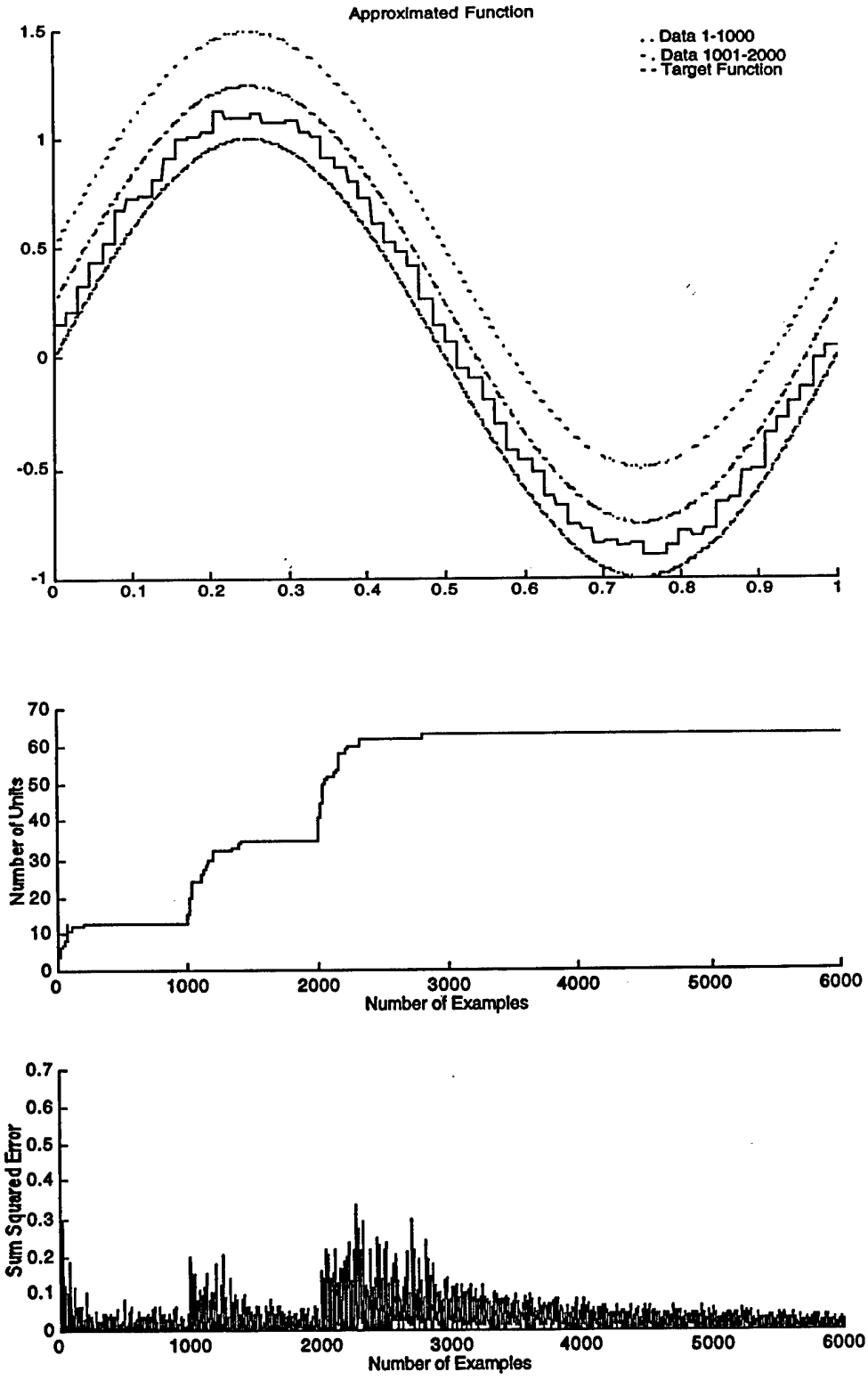


Figure 5.4 Results of Constant Value Test

Our network was able to converge to the target function. However, it needed nearly twice the amount of good data as bad data to get within the .0125 error threshold for the target function (see Figure 5.4). Another notable phenomenon was the large increase in wavelet basis units as the data sets changed. This is due to the large errors involved with the drastic changes at data points 1000 and 2000 and violates our initial assumption that errors are due to insufficient wavelet basis functions. Thus when using this algorithm with non-stationary data, another heuristic for adding units must be used.

The second example set was created by shifting the target function, $y(x) = sin(2\pi x)$:, by a constant value along the input axis:

Data Points 1-1000 : $\sin(2\pi\ (x - .2))$
Data Points 1001-2000 : $\sin(2\pi\ (x - .1))$
Data Points 2001- ... : $\sin(2\pi x)$ (the target function)

We will refer to this as the *shifted data test*. This set should be a little more difficult for the network to approximate because now not only do the coefficients need to be adjusted, but also the units responsible for each portion of the function will change.

Our network was also able to converge to the target function of this example set. It exhibited the same jumps in unit additions as the previous set. Again, this is due to the large errors associated with the drastic changes in the relative target functions. Finally, this example set

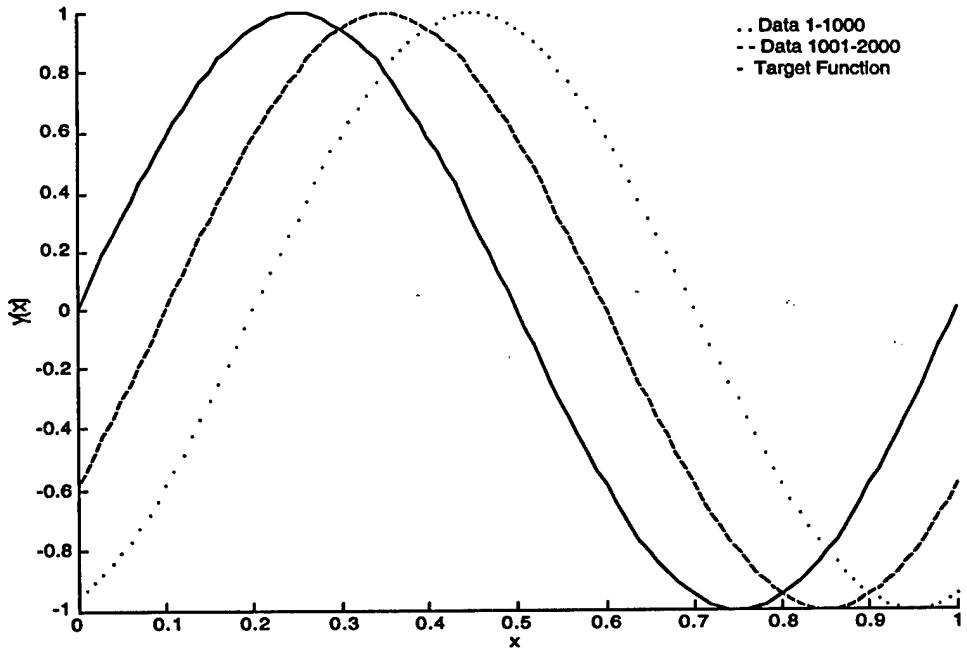


Figure 5.5 Non-stationary Data Used for Shifted Data Test

required even more good examples (approximately 6000, or three times as many) to get below the .0125 error threshold (see Figure 5.6). This is most likely due to the more difficult nature of the data set.

Figures 5.4 and 5.6 clearly show the approximation approaching the target as more good data is presented to it. If our error thresholds were set lower, the approximations would have been even closer to the targets (and the amount of good data required would have increased). However, these results are good enough to show that the network will remain stable and converge to the target values when experiencing non-stationary data.

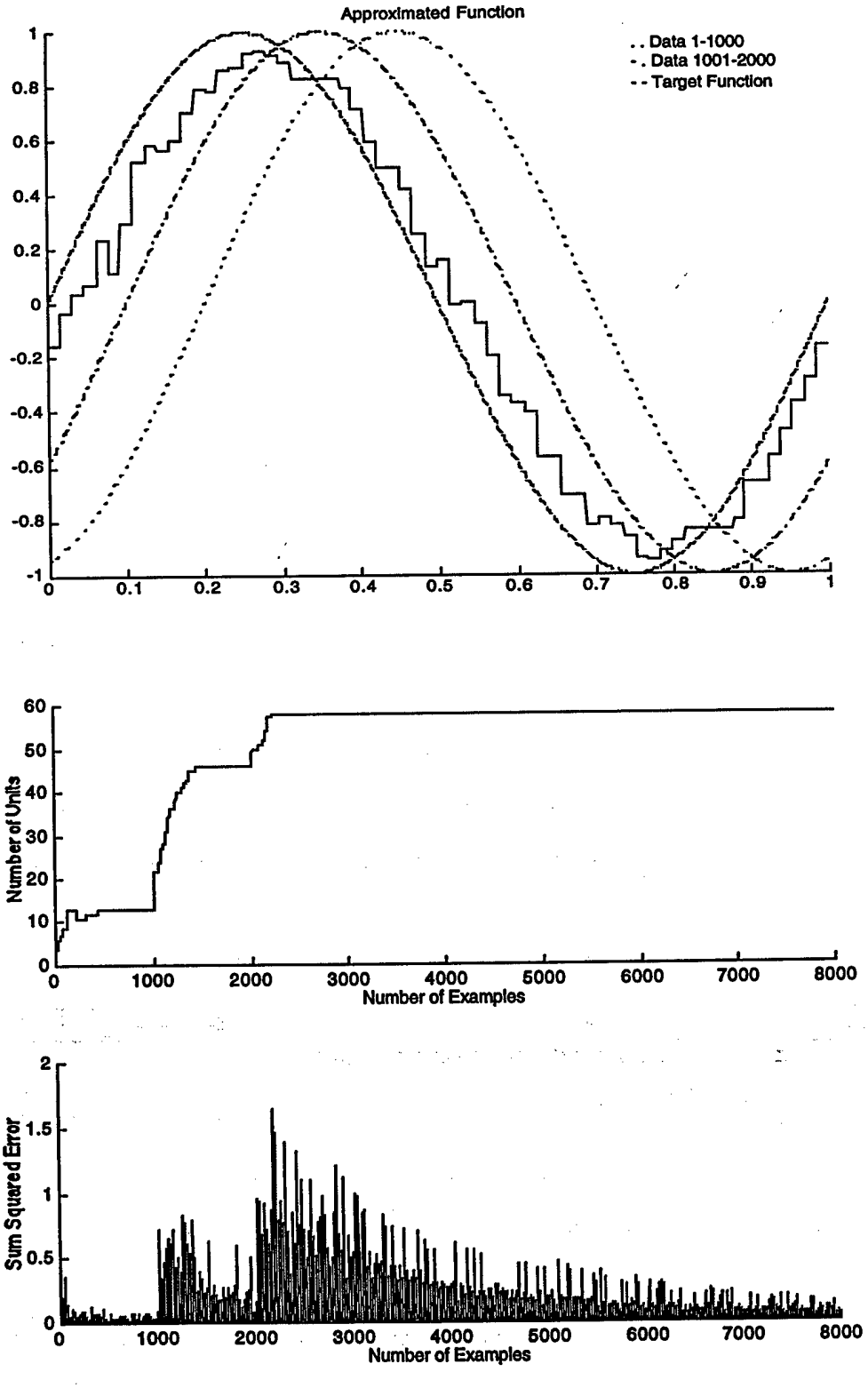


Figure 5.6 Results of Shifted Data Test

5.3.2 The Reinforcement Learning Problem

In this section we are ready to test our network with a real reinforcement learning problem. To do this we used the puck on the hill problem presented in section 5.1. However, we modified the problem somewhat to reduce the complexity and to facilitate the use of Q-learning.

First, we reduced the action set to bang-bang control, allowing only a maximum force forwards (1) and a maximum force backwards (-1). Then we discretized the state space, allowing 81 states for position and 21 for velocity. This makes Q-learning feasible by making it possible to find the function maximum easily. The problem description remains the same.

The network used for the problem will be a two input, two output Haar wavelet network. The inputs are position and velocity, and the outputs are the Q-values for each allowable action (forward or backward). Simply put, the two outputs

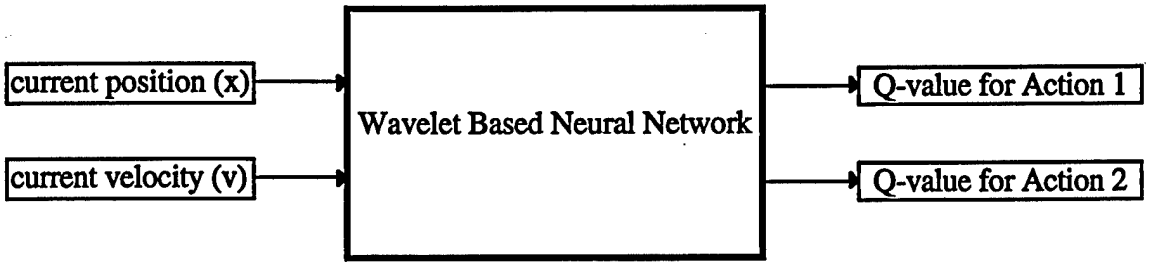


Figure 5.8 Puck on the Hill Network Layout

tell the controller the quality of each action, so it can determine which action for a given state will bring it closer to the goal. The algorithm for this problem is as follows:

- Gamma ← 0.9
- repeat
  - Generate a random *state*
  - Generate a random *action*
  - *next_state* ← transition_function(*state*, *action*)
  - if *next_state* = Goal_State
    reinforcement ← 30
    else
    reinforcement ← -1
  - *current_q* ← feedforward_net(*state*, *action*)
  - *next_q* ←
    max(feedforward_net(*next_state*, {actions}))
  - *update* ← *current_q* + Gamma ·
    (*reinforcement* + *next_q* - *current_q*)
  - train_net(*state*, *action*, *update*)

where

feedforward_net is a function that finds the network output for a given state-action pair

update is the Q-learning update that the network will be trained upon

train_net is a function that trains the network coefficients to better represent the data

5.3.2.1 Results of the Puck on the Hill Problem

The target functions for our discretized version of the puck on the hill problem are shown in Figures 5.7 and 5.8. The solution of our reinforcement learning problem does not require us to learn the target functions exactly. They simply need to be learned well enough that the network can determine the correct action for every state. In our test, we used a simulation to determine if the approximations were learned well enough. The simulation was conducted by choosing a random initial state. The network was then tasked to reach the goal state using 16 actions or less. When the

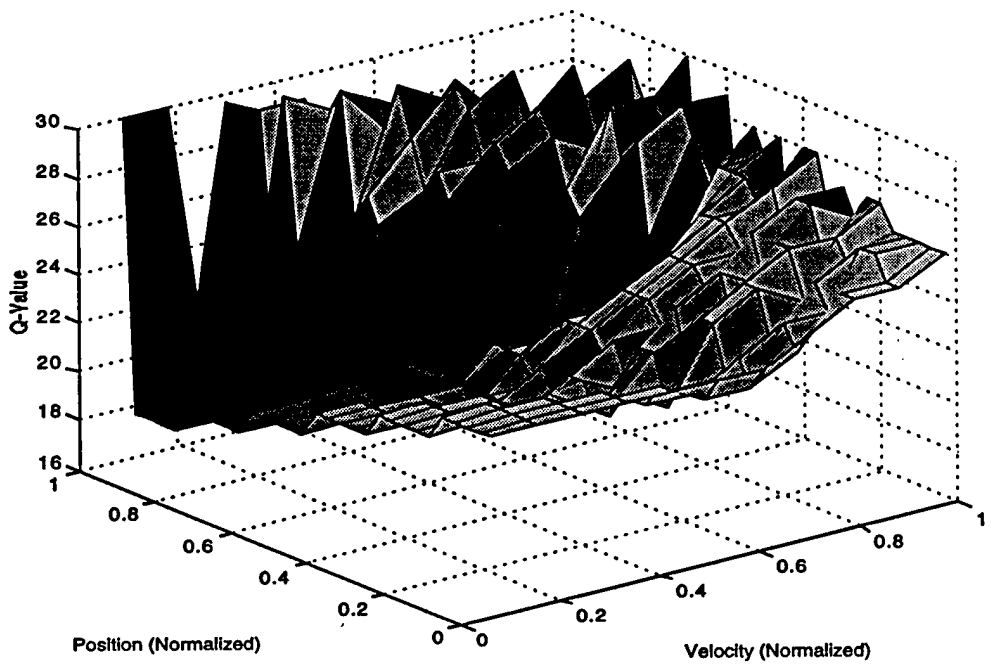


Figure 5.7 Target Function for Action # 1

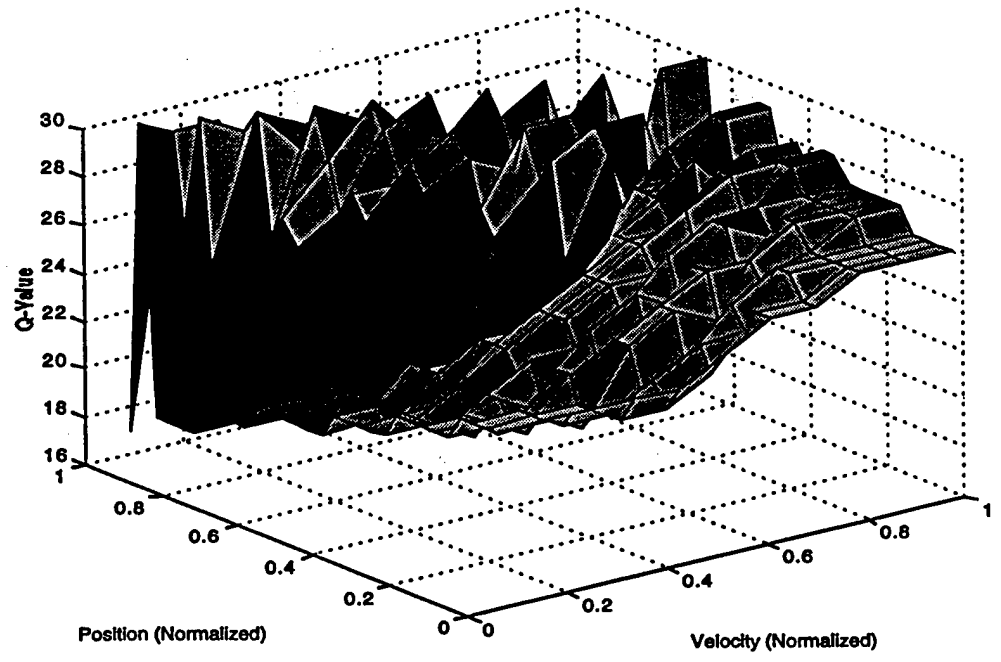


Figure 5.8 Target Function for Action # 2

network is able to reach the goal state using ten random starting points, training ends.

Our network needed 8000 epochs and 45 wavelet basis units to succeed in the simulation. Like our experiment in Section 5.2, the network was exceedingly slow, taking 1.01 hours to solve the problem (approx. 2.2 data vectors per second). However, this is probably due much more to the type of problem (many queries to the network have to be made for each epoch) than to the dimension of the data. The plots of the output approximations can be found in Figures 5.9 and 5.10. While they are not perfect approximations, they are good enough to allow the controller to make the correct decisions. Figure 5.11 shows the number of basis functions used by the network as a function of epoch.

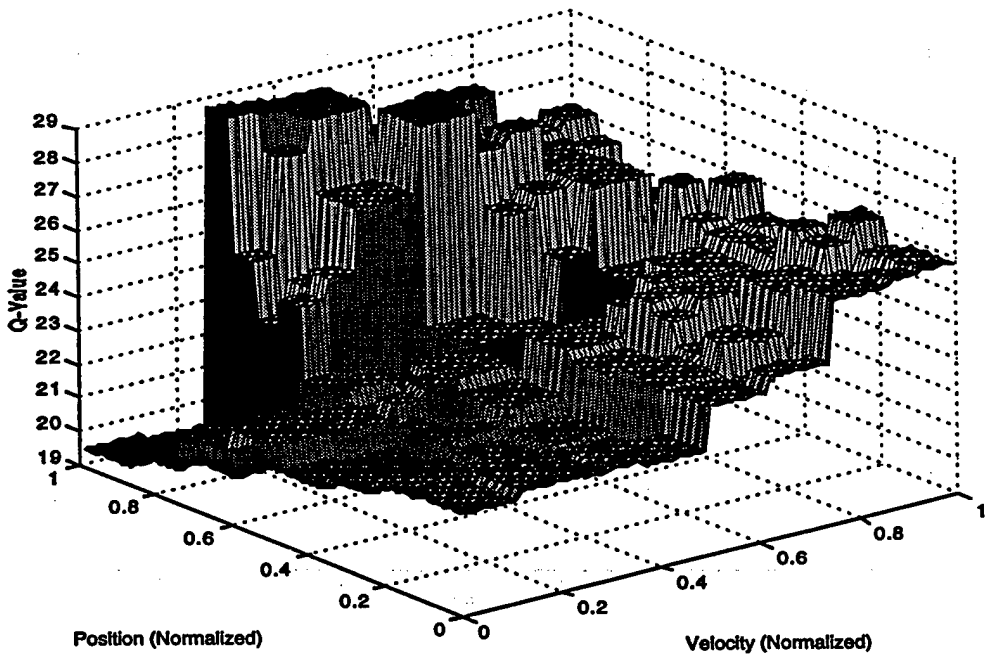


Figure 5.9 Approximated Function for Action # 1

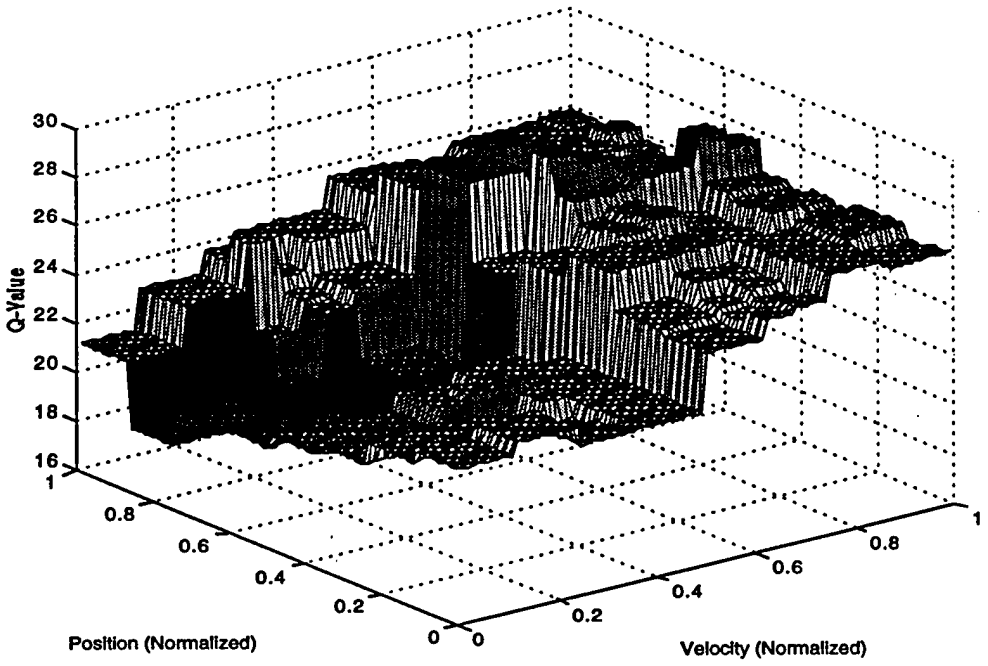


Figure 5.10 Approximated Function for Action # 2

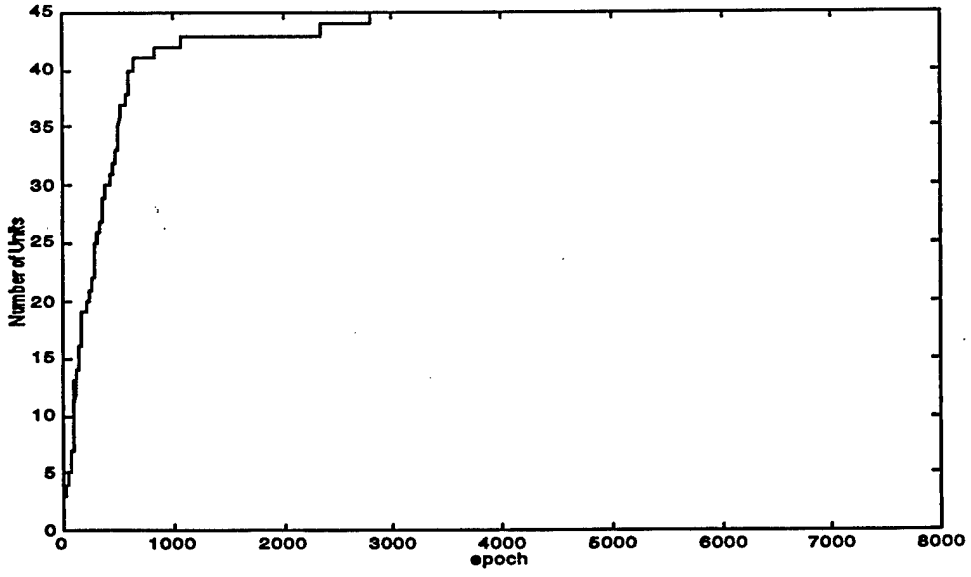


Figure 5.11 Unit Count for Puck on the Hill Problem

6 Summary

6.1 Conclusions

The motivation for this thesis was to develop a compact network structure that could be used with on-line applications while being fast as well as computationally and spatially efficient. On-line training was taken care of by the Recursive Least Squares algorithm, which also provided an efficient way to process input data. Variable structure and multiresolution provide the compact network structure. Haar wavelets and de-coupled Recursive Least Squares were used to make things faster and simpler. How well these components worked together is shown by both the successes and failures of the network on the testing material.

6.1.1 Overall Results

With all things considered, the algorithm performed well in its trials. The success of the algorithm with variable structure was very pleasing. Our algorithm not only has the ability grow to match the needs of the input data, it is able to perform that growth based upon desired error thresholds. The examples in Chapters 2 and 3 clearly show the different approximations determined by different error thresholds. The algorithm also met our criterion for a fast algorithm, at least at lower input dimensions. At input dimensions one and two, the network was able to train on 15-30 data vectors per second, *with only one pass through the data necessary.*

Simple, orthogonal wavelets and RLS takes much of the credit for that. So with these things in mind, it is reasonable to declare that this algorithm met its expectations. However, it is not without its faults.

There is no doubt that we also experienced difficulties with this algorithm. Perhaps the largest of these was the significant drop in performance with higher dimensional data. While not expressed in the requirements, there was the implicit hope that the algorithm would perform equally well on higher dimensional inputs. As seen in the Section 5.2 model approximation, the curse of dimensionality does take its toll on the algorithm. The three input, two output network was much slower than anticipated. However, the network result was still accurate and still had the good generalization that was displayed in our lower dimensional examples. Future work resulting in better variable structure techniques and different wavelet basis functions may alleviate this problem.

While Chapter 5 showed that our algorithm is capable of learning non-stationary data, it showed that it did not have a proclivity for it. This is most likely caused by the weighting used in the training algorithm (RLS) along with the additive network structure. The simple tests along with the length of time required to learn the value functions for a relatively simple reinforcement learning problem indicate that the algorithm is probably most useful in supervised learning problems with stationary data. While these problems

were unforeseen, they do not invalidate the success the algorithm displayed in our other examples. They are best regarded as important knowledge on the limitations of our algorithm.

6.1.2 Final Assessment

I believe that we accomplished in this thesis what we set out for. While the network's performance is not extraordinary, its novel design and components are definitely useful. The network performs the way it was designed to, the next step is to use some of the elements in the next section to improve upon it.

At the very least, portions of this network will be useful in the future designs of other networks. There is an undeniable need for this sort of orthogonal, variable structure framework. There is no doubt that this is not the last paper on this topic; hopefully parts of it will influence other solutions.

6.2 Recommendations for Future Research

6.2.1 High Dimensional Data

- Mode Determination. One of the most detrimental aspects of the "curse of dimensionality" for this algorithm is the 2^n-1 modes which make up the orientations of each wavelet in the n-dimensional space. It is reasonable to assume that the wavelets will only need a small number of those orientations to accurately approximate the input data. A clever way is needed to determine which orientations are necessary. This will reduce the size of the network and make it much more efficient.

- Non-separable Wavelets. These wavelets deal with the same problem discussed above, wavelet modes. In this thesis we used separable wavelets, meaning that they were built using the tensor products of one-dimensional wavelets [8, 14]. Non-separable wavelets eliminate the need for wavelet modes altogether. These wavelets are more complex than the ones used in this thesis, but in multi-dimensional space they save 2^n-1 coefficients per wavelet.

- Encoding of Frequency Data. A better way to encode frequency data must be found. In high dimensional networks the amount of frequency data that must be kept becomes very large. Currently we encode the information as bits, limiting the amount of information able to be kept for each wavelet to the maximum integer size of the machine. On our platform this limits the allowable input dimension to 5. One possible way to encode the data is to use an additional n-dimensional matrix for each wavelet unit, but this becomes a storage concern. The use of Huffman encoding or a similar, more efficient way to represent bits would be a better choice.

6.2.2 Non-Stationary and Adverse Data

- Removal of wavelets. In the case of changing, or non-stationary data there may come a time when wavelets that were once necessary for the approximation are no longer needed. Our algorithm has no means, save pruning, to remove wavelets intelligently. One easy option is to remove the top resolution levels at various times, forcing the algorithm to grow according to the new data. However, wavelets that are still useful will be removed this way, significantly affecting performance. Another idea is to keep track of the last time a wavelet basis function was hit by data, and prune according to this value. A good basis function removal heuristic would increase the applicability of our algorithm.

- Improved age weighting. The RLS age weighting discussed in section 2.3.3.1 allows us to use a discounted age weighting scheme in RLS. However, after a short time, this weighting converges to constant steady-state values. In applications such as reinforcement learning, the new data must be weighted much more heavily than older data since it contains better information. A better weighting scheme could allow this algorithm and RLS to be much more amenable to reinforcement learning applications.

- Improved inactive wavelets. Inactive wavelets are useful for implementing the spirit of this algorithm which is to

place the correct size wavelet for the given data. Unfortunately, our multiresolution structure and the data that we need to keep for each wavelet only allows us to skip one resolution at a time (placing a unit a quarter of the size of the currently smallest wavelet). A better algorithm would allow any size wavelet to be placed above the current wavelet. However, this becomes a problem if more data arrives later, necessitating a larger wavelet which could make the previous addition of the smaller wavelet superfluous. A wavelet removal algorithm could be used in conjunction to prevent this problem.

Appendix A Least Squares and the Discrete Wavelet Transform

The purpose of this appendix is to show mathematically that the Discrete Wavelet Transform is equivalent to Least Squares under these conditions:

Assumptions:

- Orthonormal basis (n basis functions)
- The Basis is covered by data that is finite, discrete, sufficiently dense and uniformly distributed (k data points)
- Training examples are supervised, i.e., an example looks like (z_j, b_j), where z_j = j-th input and b_j = j-th target value

Using Equation 2.3 from Section 2.1.1, the Continuous Wavelet Transform for an orthonormal basis is:

$$x^i = \int_{-\infty}^{\infty} a^i(z)\, b\,(z)\, dz \qquad \text{(A.1)}$$

where

$a^i(z)$ is the i-th basis function

$b(z)$ is the target function

x^i is the coefficient of the i-th wavelet

Since we are using discrete data, Equation A.1 must be modified to give us the **Discrete Wavelet Transform**, which takes the form

$$x_k^i = K_k^i \sum_{m=1}^{k} a^i(z_m)\, b_m(z_m) \qquad \text{(A.2)}$$

where

K_k^i is a normalization factor necessary to account for the loss of normality during sampling of the basis function $a^i(z)$

Equation A.2 can also be written in vector form:

$$\underline{x}_k^i = K_k^i (\underline{a}_k^i)^T \underline{b}_k \tag{A.3}$$

where

$\underline{a}_k^i$ is Basis Function Vector such that
$\underline{a}_k^i = \left[a^i(z_1) \quad a^i(z_2) \quad \cdots \quad a^i(z_k)\right]^T$

$\underline{b}_k$ is the Data Vector for data point k such that
$\underline{b}_k = \left[b_1(z_1) \quad b_2(z_2) \quad \cdots \quad b_k(z_k)\right]^T$

$\underline{x}_k$ is the Coefficient Vector for n basis functions after k data points such that $\underline{x}_k = \left[x_k^1 \quad x_k^2 \quad \cdots \quad x_k^n\right]^T$

It is now necessary to determine what K_k^i will be in Equation A.3

Example: Assume data ($\underline{b}_k$) falls exactly on basis function ($\underline{a}_k^i$), i.e., they are equivalent. In this case, we would like to normalize Equation A.3 such that the coefficient $\underline{x}_k^i$ evaluates exactly to 1:

$$\underline{x}_k^i = K(\underline{a}_k^i)^T \underline{b}_k = K(\underline{a}_k^i)^T \underline{a}_k = 1 \quad \rightarrow \quad K = \frac{1}{(\underline{a}_k^i)^T \underline{a}_k} \tag{A.4}$$

$$\therefore \quad x_k^i = \frac{(\underline{a}_k^i)^T \underline{b}_k}{(\underline{a}_k^i)^T \underline{a}_k^i} \tag{A.5}$$

For **Least Squares** we want to find the solution, x, to the system such that $\|Ax - b\|^2$ is minimized. For an overdetermined system, the solution is:

$$\underline{x}_k = \left(A_k^T A_k\right)^{-1} A_k^T \underline{b}_k \tag{A.6}$$

where

A_k is the Basis Function Matrix such that $A_k = \left[\underline{a}_k^1 \quad \underline{a}_k^2 \quad \cdots \quad \underline{a}_k^n\right]$

If we expand Equation A.6, we get:

$$A_k^T A_k = \begin{bmatrix} (\underline{a}_k^1)^T \\ (\underline{a}_k^2)^T \\ \vdots \\ (\underline{a}_k^n)^T \end{bmatrix} \begin{bmatrix} \underline{a}_k^1 & \underline{a}_k^2 & \cdots & \underline{a}_k^n \end{bmatrix} = \begin{bmatrix} (\underline{a}_k^1)^T \underline{a}_k^1 & \cdots & (\underline{a}_k^1)^T \underline{a}_k^n \\ \vdots & \ddots & \vdots \\ (\underline{a}_k^n)^T \underline{a}_k^1 & \cdots & (\underline{a}_k^n)^T \underline{a}_k^n \end{bmatrix} \qquad \text{(A.7)}$$

From Section 2.1.2, Equation 2.5 we see that if $\underline{a}_k^p$ and $\underline{a}_k^q$ are orthogonal, then:

$$(\underline{a}_k^p)^T \underline{a}_k^q = 0 \quad \text{for } p \neq q \qquad \text{(A.8)}$$

(i.e., orthogonal basis functions have inner products that are zero).

Using Equation A.8 with Equation A.7 gives us:

$$A_k^T A_k = \begin{bmatrix} (\underline{a}_k^1)^T \underline{a}_k^1 & & 0 \\ & \ddots & \\ 0 & & (\underline{a}_k^n)^T \underline{a}_k^n \end{bmatrix} \qquad \text{(A.9)}$$

Equation A.9 is now very easy to invert, becoming:

$$\left(A_k^T A_k\right)^{-1} = \begin{bmatrix} \dfrac{1}{(\underline{a}_k^1)^T \underline{a}_k^1} & & 0 \\ & \ddots & \\ 0 & & \dfrac{1}{(\underline{a}_k^n)^T \underline{a}_k^n} \end{bmatrix} \qquad \text{(A.10)}$$

Plugging Equation A.10 into Equation A.6 gives us:

$$\underline{x}_k = \begin{bmatrix} \dfrac{1}{(\underline{a}_k^1)^T \underline{a}_k^1} & & 0 \\ & \ddots & \\ 0 & & \dfrac{1}{(\underline{a}_k^n)^T \underline{a}_k^n} \end{bmatrix} \begin{bmatrix} (\underline{a}_k^1)^T \\ (\underline{a}_k^2)^T \\ \vdots \\ (\underline{a}_k^n)^T \end{bmatrix} \underline{b}_k = \begin{bmatrix} \dfrac{(\underline{a}_k^1)^T \underline{b}_k}{(\underline{a}_k^1)^T \underline{a}_k^1} \\ \vdots \\ \dfrac{(\underline{a}_k^n)^T \underline{b}_k}{(\underline{a}_k^n)^T \underline{a}_k^n} \end{bmatrix} \qquad \text{(A.11)}$$

From Equation A.11 we can now decouple the coefficients for individual, orthogonal basis units:

$$x_k^i = \frac{(a_k^i)^T b_k}{(a_k^i)^T a_k^i} \qquad \text{(A.12)}$$

Equation A.12, the Least Squares solution, is identical to Equation A.5, the Discrete Wavelet Transform Solution!

Therefore, **Least Squares ≡ Discrete Wavelet Transform under our assumptions.**

References

[1] Bakshi, Bhavik R., Alexandros Koulouris, and George Stephanopoulos (1994). "Wave-Nets: Novel Learning Techniques, and the Induction of Physically Interpretable Models", SPIE, Vol. 2242, pp. 637-648.

[2] Bakshi, Bhavik R., Alexandros Koulouris, and George Stephanopoulos (1995). "Empirical Learning Through Neural Networks: The Wave-Net Solution", Advances in Chemical Engineering, Vol. 22, pp. 437-483.

[3] Cybenko, G. (1989). "Approximation by Superpositions of a Sigmoidal Function", Math. Control Signal Systems, Vol. 2, pp. 303-308.

[4] Daubechies, I. (1992). *Ten Lectures on Wavelets*, SIAM.

[5] Daubechies, I. (1988). "Orthonormal Bases of Compactly Supported Wavelets", Communications on Pure and Applied Mathematics, Vol. XLI, pp. 909-996.

[6] Hubbard, Barbara Burke (1996). *The World According to Wavelets*, A K Peters.

[7] Kovacevic, J. C., and M. Vetterli (1992). "Non-separable Multi-Dimensional Perfect Reconstruction Filter Banks and Wavelet Bases for R^n", IEEE Transactions on Information Theory, Vol. 38, Number 2, pp. 533-542.

[8] LiMin, Fu (1994). *Neural Networks and Computer Intelligence*, McGraw-Hill.

[9] Livstone, Mitchell M. (1994). *Wavelets: A Conceptual Overview*, CSDL Report R-2602.

[10] Mallat, Stephane G. (1989). "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 11, Number 7, pp. 674-693.

[11] Muller, Peter and Briani Vidakovic (1991). "Wavelets for Kids", Institute of Statistics and Decision Sciences, Duke University. pp. 1-26.

[12] Peng, Jing and Ronald Williams (1993). "Efficient Learning and Planning Within the Dyan Framework", Adaptive Behavior, Vol. 1, Number 4, pp. 437-454.

[13] Russell, Stuart and Peter Norvig (1995). *Artificial Intelligence: A Modern Approach*, Prentice Hall.

[14] Strang, Gilbert (1986). *Introduction to Applied Mathematics*, Wellesley-Cambridge Press.

[15] Sutton, R. S. (1990). "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming", *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann.

[16] Zhang, Qinghua and Albert Benveniste (1992). "Wavelet Networks", IEEE Transactions on Neural Networks, Vol. 3, No. 6, pp. 889-898.